Regular Expression
yet another introduction

正则指引（第2版）

余晟 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

本书综合作者自己遇到的实际问题，以及其他开发人员咨询的问题，总结出一套巧妙运用正则表达式的办法，并通过具体的例子指导读者拆解、分析问题。全书分为三部分：第一部分主要讲解正则表达式的基础知识，涵盖了正则表达式中常见的各种功能和结构；第二部分主要讲解关于正则表达式的更深入的知识，详细探讨了编码问题、匹配原理、解题思路；第三部分将之前介绍的各种知识落实到常用语言.NET、Java、JavaScript、PHP、Python、Ruby、Objective-C、Golang 中，在详细介绍了在这些语言中正则表达式的具体用法之外，还辨析了版本之间的细微差异。本书既可以作为专门的学习用书，也可以作为备查的参考手册。

本书适合经常需要进行文本处理（比如日志分析或网络运维）的技术人员、熟悉常用开发语言的程序员，以及已经对正则表达式有一定了解的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

正则指引/余晟著. —2 版. —北京：电子工业出版社，2018.11
ISBN 978-7-121-35130-3

Ⅰ. ①正… Ⅱ. ①余… Ⅲ. ①正则表达式 Ⅳ.①TP301.2

中国版本图书馆 CIP 数据核字（2018）第 224858 号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：24.5　字数：577 千字
版　　次：2012 年 5 月第 1 版
　　2018 年 11 月第 2 版
印　　次：2018 年 11 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

引子：关于正则表达式……

正则表达式这个名字看起来总有点古怪，概念似乎也不简单，甚至需要用一整本书来讲解。可是，它到底是什么呢？

同为技术人员，我相信你总会与字符串打交道，相应的，各种语言也都提供了与字符串有关的函数。我们先看下面几个问题，用字符串函数是如何解决的（下面的代码使用 Python 语言，它很直观，正文里有基础的介绍。现在，你只需要知道 def 是定义函数的关键词即可）。

引入正则表达式

1. 判断字符 ch 是否是数字字符

```
def isDigit(ch) :
    return ch == "0" or ch == "1" …… or ch == "9"
```

2. 判断字符串 str 是否是电话号码（为简单起见，现在只考虑固定电话号码，也就是长度在 7~8 位之间的数字字符串，且第一位不为 0）

```
def isPhoneNum (str) :
    if len(str) >= 7 and len(str) <= 8 and str[0] != "0" :
        for ch in str :
            if not isDigit(ch) :
                return false
        return true
    return false
```

任务的复杂度并没有增加太多，程序的复杂度增加了很多倍；如果你不同意，那么，来一个更复杂的。

3. 找出一段文本中所有的电话号码

最直接的办法是，在字符串中的每个位置截取 7~8 个字符，调用之前的 `isPhoneNum()`。这么做看起来没问题，只是效率太低。

当然，做点改进也不难，加上一个前置条件，只在“当前字符为数字字符”的情况下调用

isPhoneNum()。这样效率倒是改进了，但是还有问题没有解决：要求找到的是长度大于等于 7 个字符、小于等于 8 个字符的“数字字符串”，而不是“子字符串”——也就是说，假如数字字符串是 64240000，需要将它找出来；如果数字字符串是 13800138000，则需要忽略它，以及其中的任何子串（比如 13800138、00138000）。

所以，用 isPhoneNum()找出字符串之后，还需要保证它之前的字符不是数字字符，之后的字符也不是数字字符。看起来很简单，但合格的程序员一定要考虑边界问题，避免越界错误：如果当前字符是整段文本的第一个字符，则不需要判断之前的字符，因为它不存在；同样，如果找出的字符串在整段文本的末尾，则不需要判断之后的字符，因为它同样不存在……

到现在为止，即便只是找到最简单的固定电话号码，程序也非常复杂，难以维护。如果要查找的是形式更多变的文本，比如带区号的电话号码（021-64240000 或者 03718888888）、手机号码（13800138000、+8613800138000 或者 013800138000），程序更是不可想象，更不用说文件路径名、URL 地址、电子邮件地址了！然而，日常开发中我们又确实经常需要面对这类任务，有什么更好的办法呢？

正则表达式就是解决这类问题的万能药。虽然许多人有点看不起它，觉得不入流，一些科班教材里也不会花太多篇幅来介绍它，但它确实是解决问题的利器——之前提到的三个例子，用正则表达式都可以轻松解决。

引入正则表达式之后

1. 判断字符 ch 是否是数字字符

```
def isDigit(ch) :
    return re.search(ch, "[0-9]") != None
```

看起来很复杂，其实并不复杂：这里真正要关心的就是正则表达式`[0-9]`，它表示“从 0 到 9 之间的任意字符”，很形象吧？re.search()是正则表达式运算函数，它判断 ch 能否由正则表达式[0-9]匹配，可以则返回一个结果，否则返回 None（这些细节正文中会讲到）。

2. 判断字符串 str 是否是电话号码

```
def isPhoneNum(str) :
    return re.search(str, "[1-9][0-9]{6,7}") != None
```

这个正则表达式最开始是`[1-9]`，表示第一个字符必须是 1~9 之间的数字字符；之后是`[0-9]{6,7}`，表示长度在 6 和 7 之间，由 0~9 之间的数字字符组成的字符串（两部分加起来，整个字符串的长度在 7 和 8 之间）。要解决的问题复杂了，正则表达式仍然直观形象。

3. 找出一段文本中所有的固定电话号码

```
def findNumStr(str) :
    return re.findall(str, '(?<![0-9])[1-9][0-9]{6,7}(?![0-9])')
```

这个正则表达式之前多出了`(?<![0-9])`，表示“之前不能是[0-9]”；之后多出了`(?![0-9])`，表示“之后不能是[0-9]”。虽然稍微复杂点，但意思明确，而且不难理解。`re.findall()`的意思也很明显：找到所有这样的字符串。

可以想象，循着这种思路，查找更复杂的电话号码、手机号码等任务都不难解决。更重要的是，之前需要许多行语句才能完成的任务，现在基本上只需要一个正则表达式、一条语句就可以完成。正因为如此，不少人虽然认为正则表达式不够花哨、漂亮，却不得不承认它是一种“匕首应用”——匕首，没有十八般兵刃那么气派，关键时候却不可或缺，所以值得花时间练练。同样，正则表达式虽然不能用来显摆，但总有派得上用场的地方，花时间练练绝不是坏事。即便你的工作不是纯粹的文本处理（比如日志分析），也总会有用到正则表达式的地方（比如查找和修改源代码），所以我希望，这本书能陪伴你练出一身正则表达式的好功夫，在关键场合能亮出称手的工具。

最后，为了传承经典教科书的良好习惯，附上正则表达式的“科班史”。

正则表达式发源于与计算机密切相关的两个领域：计算理论和形式语言。20 世纪 40 年代，两位神经生理学家 Warren McCulloch 和 Walter Pitts 发明了一种用数学方式来描述神经网络的办法，他们把神经系统中的神经元描述成小而简单的自动控制单元。1956 年，数学家 Stephen Cole Kleene 在他们研究的基础上，发表了一篇名为《神经网事件的表示法》的论文，在其中，他采用了一些称之为“正则集合（regular set）”的数学符号来描述神经网络模型。

之后，UNIX 的主要发明人 Ken Thompson 将这个符号系统引入了文本编辑器 QED（意思是“在文本中搜索某种模式”），正则表达式由此也进入了计算机世界。随后 Ken Thompson 又将正则表达式引入了 UNIX 下的文本编辑器 ed，ed 最终演化为大家熟悉的 grep（grep 得名自 ed 编辑器中的正则表达式搜索命令 g/re/p，其中的 re 表示“正则表达式”）。

返璞归真——评《正则指引》

第一次接触正则表达式，是 2000 年我在西安一家公司使用 Perl 做网站开发时。之前我在工作中只使用过标准的 C 语言，Perl 这门编程语言的强大表达能力，令我印象极为深刻。Perl 的力量，除了语言本身的设计之外，很大程度上来自它对正则表达式的完美支持。当时我们开发了一个网上商城的应用，允许很多商家在这里开店，可以选择一些不同的样式模板。我很快发现，使用 Perl+正则表达式是开发这类应用的利器。我们只花了大约一个月的时间，就完成了网站核心功能的开发。那时候我意识到，使用正则表达式是聪明人写程序的方法（没说我是聪明人，但是我非常希望与那些聪明人为伍），可以极大地提高代码的重用度和执行效率。如果完全不使用正则表达式，代码量会增加数倍甚至数十倍。

后来因为一些原因，我告别了 Perl。在之后的工作中，我使用过 Java、JavaScript、Ruby 等编程语言。我发现这些语言对于正则表达式的支持，没有一个能够超越 Perl。Java 这种所谓的“工业主流编程语言”，一直到 2002 年 JDK 1.4 推出时，才正式把对正则表达式的支持加入核心类库。因为长期缺乏对正则表达式的原生支持，以及语言本身表达能力欠缺，使用 Java 来做大量的文本处理，感觉非常笨拙，完全没有使用 Perl 那种指哪打哪的快感。直到 2007 年我发现了另一个更好的 Perl 语言——Ruby，才重新找回了 2000 年 Perl 带给我的编程快感。

因为我的工作主要是做 Web 开发，大量的时间花在与 HTML/CSS/JavaScript 以及关系数据库打交道上。在这里并没有很高深的算法，只有大量繁重的文本处理。难以想象，如果没有正则表达式，我们的开发将会是何等原始。

除了 Web 开发领域，需要实现大量自动化功能的一些领域，例如运维领域和自动化测试领域，也是正则表达式大显身手的地方。无论使用稍显简陋的 sed/awk 还是更高级的 Perl/Python/Ruby，实现自动化功能，都必须依赖大量的正则表达式。

自从面向对象的编辑方式时髦起来之后，甚至一度出现了面向对象万能论，有人试图用 MDA 和可执行的 UML 来解决一切编程问题。但是我一直认为面向对象只解决了软件开发的一小部分问题，而且是宏观方面的问题。正则表达式解决的问题，是面向对象无能为力的一些微观方面的问题。在这里不需要坐而论道的方法论争论，需要的是刺刀见红的肉搏战。这些问题即使使用完全面向对象的方式能够解决，也会是很笨拙的。如果用物理学来比喻，面向对象是“广义相对论”，

而正则表达式则是“量子力学”。

正则表达式已经成为现代编程语言的基础模块，现在很难找到一种不支持正则表达式的编程语言。除了编程语言外，在很多工具软件，例如文本编辑器（Vi、Emacs、UltraEdit）、Web 服务器（Apache、Nginx）中都能找到正则表达式的身影。

余晟老师是我的朋友，我对他印象最为深刻的是他对于技术工作的严谨态度。“格物致知”是中国传统儒家学派所追求的一种道德修养，也是一种境界。余老师是我的朋友中最接近“格物致知”这种境界的一位。我虽然从未精通过任何一门技术，但是很喜欢结交余老师这样的朋友。

余老师潜心编著的这本《正则指引》深入浅出，将正则表达式的由来和分支娓娓道来。阅读这本书，我仿佛回到了 11 年前做 Perl 程序员时的快乐时光。国内很多程序员的一个通病是好高骛远，像《正则指引》这样一本详细讲解基础知识的书未必会有很好的销路，但是等你做过很多年开发之后，你会发现，对你最有价值的，正是这些基础知识和工具。软件开发的“道”，正是隐藏在这些看起来不起眼的基础知识和工具之中的。

李锟

2011 年 11 月 25 日

克制我们内心的冲动

《正则指引》（第 2 版）就要出版了，按说这是一条好消息。在这条好消息面前，我更想做的是克制自己内心的冲动，静下心来讲讲这本书初版以来的故事。

《正则指引》初级刚出版的时候，我一度认为，自己和正则表达式的缘分到此为止了。如果说翻译《精通正则表达式》之后还有许多遗憾，比如某些讲解方式不符合中国程序员的思维，以及过份关注英文，所以关于东亚文字处理的知识无从寻找，经验无从分享。那么写作《正则指引》，就是弥补这种缺憾的绝好机会。《正则指引》面世之后，这些缺憾已经悉数补上了。

令我没有想到的是，《正则指引》自 2012 年出版以来，不断有读者向我反馈问题。除去最早一两年密集热烈的反馈，后续的反馈如涓涓细流绵绵不绝。而我一度想当然地认为勘误已经完整了，所以一直没检查勘误邮箱。直到一年前读者在微信公众号后台给我留言，详细列明勘误意见之外，毫不留情地指责“对自己的作品不负责，长期不回复读者意见”。这封信让我惭愧不已。在软件开发中，“发布了就不管”是很不负责的，在技术书籍的写作中，“出版了两年就不回复读者意见”，同样是很不负责的。

所以，我必须克制自己内心“年代久远，不值得继续打理”的偷懒冲动。文责自负，完整的说法应该是“文责终身自负”。

本次《正则指引》（第 2 版）的出版，对我而言是全新的补过机会，可以“一次性”回复迄今为止所有的读者意见。当然，新增的 Objective-C、Golang 等章节，尽管已经找熟悉的朋友审读过，但我可以肯定，它们必定不是完美无瑕的，未来仍然需要坦然面对广大读者持续的批评指正。哪怕有些批评指正的语气不那么让人舒服，甚至“看不到几分善意”，我仍然需要克制自己内心“反唇相讥”的冲动，认清事实，撇开情绪，虚心面对。

同时，我也希望读者在阅读这本书时，能克制自己内心的冲动。

我希望大家克制的第一重冲动，是浅尝辄止——“正则表达式这玩意儿，要用时翻翻就好，没必要深究”。正则表达式已经诞生很多年了，以今天的标准来看，它的语法和结构相当粗陋，不幸的是，它的内部逻辑又相当复杂。有些朋友会问我一些“怎么看也看不懂”的正则表达式，坦白地说，我也要反复琢磨才能看懂。所以，尽管这本书提供了若干“速查”资料，但我还是建

议读者能耐下心来，至少通读一遍。正则表达式有点像游泳，学会了就不会忘，用的时候自然能想起来。否则，你永远只能在岸边扑腾，离开了其他人的协助，一步都不敢往深处去。虽然很多时候，与你要的东西就只有一步之遥。

我希望大家克制的第二重冲动，是玩弄正则表达式的快感。前面说过，正则表达式的语法和结构相当粗陋，内部逻辑又相当复杂，所以不少人学会之后，产生了“掌握神奇魔咒”的快感。凡是和字符串相关的处理必亮出神奇的正则表达式，能用一个正则表达式的绝不用两个，能用高级特性的绝不用简单特性……随之而来的是其他人查错时层出不穷的抱怨，更不用提更新时胆战心惊的烦恼。要知道，熟练使用正则表达式，却不滥用正则表达式，同时考虑合作同事的感受和效率，才能真正赢得大家的尊敬。

武学大师说：武功不是用来伤害，而是用来制止伤害的。哲学大师说：没有审慎思考，不懂得克制的人生，是不值得过的。这些道理听起来有悖常理，我花了不少时间才终于弄懂。我相信，正在阅读这本书的你，也应当懂得这些道理。

特别感谢两位女士，西乔和刘舫。西乔为这本书设计的封面，丝毫不受岁月的影响。刘舫编辑细致严谨的工作态度，支撑着我完成《正则指引》的第 2 版，再写下这篇序。

余晟

2018年9月3日

前言

提到正则表达式，许多人很有点不屑一顾：这东西，不登大雅之堂，再说也不是总要用到，何必专门花时间学习？

没错，正则表达式并不“总要用到”，但如果到了需要的场合不会用，往往面临“一分钱难倒英雄汉”的困境。经常需要处理文本的程序员自然知道正则表达式的价值，其他的程序员如果不会正则表达式，即便开发的领域与文本处理没什么关系，也难以躲过“躺着中枪”的命运——前几天我遇到一个问题，将一行长长的地址拆分成多行，负责这部分的程序员的日常工作只是制作 PDF 而已，拆分地址是很“边缘”的功能，但不会正则表达式就无法准确折行（一般需要在标点符号出现的地方折行，而不能只在空白字符处折行，但是不同语言中的标点符号各有不同），结果一筹莫展；相反，如果了解正则表达式，就可以很容易地处理各种语言中的标点字符。

按照我的开发经验，专门花点时间学习一下正则表达式，确实很有必要。目前可以见到的关于正则表达式的书籍和资料有不少，但又各有不足。

在互联网上，流传着一些编程语言的正则文档和《30 分钟教会你正则表达式》之类的帖子。这类资料的好处是简单直接，如果有现成的例子，而且适用于自己的语言，则可以直接抄来用。然而，其坏处也是简单直接，因为缺乏背后原理的讲解，如果找不到现成的例子，或者找不到能在自己所使用语言中行得通的例子（要知道，一种语言下的正则表达式往往并不能直接套用到另一种语言中），则束手无策。

在正式的出版领域，已经有《精通正则表达式》、《正则表达式必知必会》之类的书籍出版，尤其是前者，堪称关于正则表达式的经典著作，如果想认真学习正则表达式，这类书籍是必须阅读的。但这类书籍的弱点也很明显，即都是由英文版本翻译而来的，更多侧重英文文本的处理，身为中文世界的开发人员，我们经常需要处理中文文本——英文之外的字符。其实对于非英文字符的处理，正则表达式已经提供了足够丰富的功能，可惜资料相当匮乏。

为解决这些问题，我花了很多时间研习各种资料，然后经常给人讲解正则表达式的相关知识。我发现，很多人并不是不努力学，实在是合适的资料太少了。所以，我斗胆写作这本书。

相对于正则文档和速成教学帖子，本书深入讲解了匹配背后的原理，而且往往会举一反三，

告诉读者，这里为何这样写，如果改成其他形式，会造成什么结构差异；同时集中讲解和比较了多种语言中正则表达式用法的异同，方便读者把现成的正则表达式“移植”到自己的工作环境中。

相对于《精通正则表达式》等“正式”的书籍，本书辟出专门的章节讲解语言和编码，告诉读者如何设定编码，如何正确处理中文字符等。另外，本书还涵盖了.NET、Java、JavaScript、PHP、Python、Ruby、Objective-C、Golang 等常用语言，为每种语言专门撰写相关内容，不但详细介绍了语言中正则表达式的用法，更辨析了版本之间的细微差异，既可以作为专门学习的教材，也可以成为有用的参考手册。

本书结构

本书分为三部分。

第一部分主要讲解正则表达式的基础知识，覆盖常见正则表达式中的各种功能和结构。看完前 3 章，就可以基本弄明白现在流行的各种正则表达式；如果你之前有一些经验，会觉得阅读起来并不困难。但是我也希望读者不要忽略其他的内容，断言和匹配模式现在已经是正则表达式的“标准配置”了，而且确实可以派上大用场，所以第 4 章和第 5 章的内容，即便不是很熟悉，阅读起来可能有一些麻烦，但也不应该忽略。最后的第 6 章，则厘清了正则表达式在使用中的若干疑惑，了解它们，你就可以相对自如地穿行于正则表达式的世界了。

第二部分主要讲解关于正则表达式的深层次知识，这一部分用 3 章的内容，详细探讨了编码问题、匹配原理、解题思路。这部分内容更抽象，需要多花一点时间来阅读和理解，但是它们确实可以帮你在正则表达式的世界里登堂入室，脱离“术”的层面，掌握万变不离其宗的“道”。

第三部分的作用是接地气，将之前介绍的各种知识落实到常用语言.NET、Java、JavaScript、PHP、Python、Ruby、Objective-C、Golang 中来。每一章的开头有正则功能列表，其中的功能对应着前面部分的讲解，这些功能的具体应用实例以及不同版本之间的差异，则在章节中详细讲解，每一章的最后还给出了常见任务的示例代码，方便日后查询。第 18 章简要介绍了正则表达式在 Linux 下常用工具 vi、grep、awk、sed 中的使用，并通过一个实际的例子将这几种工具串起来，对比说明了它们适合解决的问题。

在本书的最后提供了用作参考的 3 个附录。

附录 A 是正则表达式的常用功能在不同语言中的比对，希望能给需要在多种语言中使用正则表达式或者移植正则表达式的读者提供一份有用的参考；附录 B 给出了若干常见的正则表达式，比如匹配邮政编码、身份证号、手机号、QQ 号、电子邮件地址等，希望能成为常见问题的“速查手册”；附录 C 列出了常用正则表达式的工具和资源，方便大家调试自己的正则表达式，以及继续深入学习。

本书的读者对象

本书适合以下几类读者：

经常需要进行文本处理（比如日志分析或网络运维）的技术人员。这些读者或许已经熟悉了正则表达式的基本用法，但面对日益复杂化和海量的数据，阅读本书可以帮助大家更准确、更高效地处理文本，提升自己工作的价值。

熟悉常用开发语言的程序员。虽然这些读者不需要专职进行文本处理，但源代码和许多数据其实也是文本，如果不会正则表达式，在偶然遇到处理源代码或文本数据的任务时，往往会产生无力感。本书的第三部分可以帮你快速找到有关的例子，并落实在自己的编程语言中。当然前两部分也非常有必要，因为它们可以帮你夯实基础。

对正则表达式已经有一定了解的读者。这些读者虽然能用正则表达式解决常见的问题，但未必了解正则表达式的编码问题、匹配原理、解题思路，仔细阅读本书的第二部分，可以深入完善对正则表达式的理解；而第三部分详细比较了可以使用正则表达式的各种语言，以及同一种语言中各种版本的差异。所有这一切，应该可以让你对正则表达式的掌握更上一层楼。

致谢

一本书的完成，离不开众多人的帮忙。

首先要感谢的是李笑来老师、周筠老师以及徐定翔和卢鹄翔两位编辑。在我翻译完《精通正则表达式》之后，李笑来老师三番五次地鼓励我写一本关于正则表达式的书，并且打消了我的很多顾虑；周筠老师、徐定翔和卢鹄翔两位编辑在我写作的最初阶段做了大量细致耐心的工作。可以说，没有他们，我就不会有写作这本书的念头，也不会有坚持完成的动力。

然后要感谢的是电子工业出版社的杨福平副总编、张月萍编辑、张春雨编辑和刘舫编辑，没有他们的关照和辛勤工作，这本书的出版定然会遇到更多的困难。

感谢我的朋友霍炬和韩磊，虽然我之前阅读过《精通正则表达式》，但与翻译和写作结缘，他们给了我莫大的帮助，于是今天才有了《正则指引》这本书。尤其值得一提的是，霍炬的夫人西乔，精心手绘了这本书的封面，我在这里要对她表示诚挚的谢意。

感谢我曾工作过的盛大创新院以及创新院的各位同事（李骏、郝培强、庄表伟、丁宇、许式伟、莫华枫、李道兵、赵劼、樊一鹏、张一宁等），创新院给了大家宽松自由的工作环境，与各位同事的讨论加深了我对正则表达式的理解，也为我提供了许多例子。

感谢张东亮、陆亦斌、孙勇、叶劲峰等各位朋友，愿意拨冗阅读本书的草稿，并提出了大量专业的意见。

感谢何源、陈钢、贺钧、陈驰等读者，试读本书之后提出了大量的宝贵意见，在最后关头打

消了我心中的许多忐忑。

在更早之前，我的父母从小就鼓励我研究和了解各种科学原理（“玩也要动脑筋”），我之所以有兴趣探究正则表达式背后的世界，而不满足于“够用/凑合”，都是受益于这种思维行为习惯。在中小学阶段，我的语文老师罗碧玉、易玺铭培养了我对于文字的兴趣，在大学阶段，东北师范大学文学院的王确老师给了我这个理科生非常多的帮助和指引。对各位师长，在此一并表示感谢，能遇到你们是我的幸运。

最后还需要感谢许多为这本书做出过贡献的人，你们的名字我可能暂时无法记起，或者无法一一罗列，但我会在心中保持对你们的感谢。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/35130

目录

第一部分

第 1 章　字符组 2

1.1　普通字符组 2

1.2　关于 Python 的基础知识 4

1.3　普通字符组（续） 6

1.4　元字符与转义 8

1.5　排除型字符组 10

1.6　字符组简记法 12

1.7　字符组运算 14

1.8　POSIX 字符组 15

第 2 章　量词 17

2.1　一般形式 17

2.2　常用量词 19

2.3　数据提取 21

2.4　点号 23

2.5　滥用点号的问题 23

2.6　忽略优先量词 26

2.7　转义 31

第 3 章　括号 33

3.1　分组 33

3.2　多选结构 39

3.3　引用分组 44

3.3.1　反向引用 48

3.3.2　各种引用的记法 50

3.3.3 命名分组 53
3.4 非捕获分组 55
3.5 补充 56
3.5.1 转义 56
3.5.2 URL Rewrite 56
3.5.3 一个例子 58

第 4 章 断言 60
4.1 单词边界 60
4.2 行起始/结束位置 62
4.3 环视 69
4.4 补充 75
4.4.1 环视的价值 75
4.4.2 环视与分组编号 76
4.4.3 环视的支持程度 77
4.4.4 环视的组合 79
4.4.5 断言和反向引用之间的关系 81
4.4.6 逆序环视的诡异之处 81

第 5 章 匹配模式 83
5.1 不区分大小写模式与模式的指定方式 83
5.2 单行模式 86
5.3 多行模式 87
5.4 注释模式 89
5.5 补充 91
5.5.1 更多的模式 91
5.5.2 修饰符的作用范围 91
5.5.3 失效修饰符 92
5.5.4 模式与反向引用 93
5.5.5 冲突策略 93
5.5.6 哪种方式更好 94

第 6 章 其他 95
6.1 转义 95
6.1.1 字符串转义与正则转义 95
6.1.2 元字符的转义 99
6.1.3 彻底消除元字符的特殊含义 101

6.1.4 字符组中的转义 103
6.2 正则表达式的处理形式 103
6.2.1 函数式处理 104
6.2.2 面向对象式处理 104
6.2.3 比较 105
6.2.4 线程安全性 106
6.3 表达式中的优先级 108
6.4 回车和换行 109

第二部分

第 7 章 Unicode 112
7.1 基础知识 112
7.2 关于编码 115
7.3 尽量使用 Unicode 编码 116
7.4 Unicode 与字符组简记法 120
7.5 规范化问题 122
7.6 单词边界 123
7.7 码值转义序列 125
7.8 Unicode 属性 127
7.8.1 Unicode Property 128
7.8.2 Unicode Block 128
7.8.3 Unicode Script 129
7.9 Unicode 属性列表 130
7.9.1 Unicode Property 130
7.9.2 Unicode Block 131
7.9.3 Unicode Script 135
7.10 POSIX 字符组 135
7.11 Emoji 136

第 8 章 匹配原理 138
8.1 有穷自动机 138
8.2 正则表达式的匹配过程 139
8.3 回溯 142
8.4 NFA 和 DFA 144

第 9 章 常见问题的解决思路 146
9.1 关于元素的三种逻辑 146
9.1.1 必须出现 147
9.1.2 可能出现 147
9.1.3 不能出现 148
9.2 正则表达式的常见操作 150
9.2.1 提取 150
9.2.2 验证 156
9.2.3 替换 160
9.2.4 切分 165
9.3 正则表达式的优化建议 167
9.3.1 使用缓存 167
9.3.2 尽量准确地表达意图 168
9.3.3 避免重复匹配 168
9.3.4 独立出文本和锚点 169
9.4 别过分依赖正则表达式 170
9.4.1 彻底放弃字符串操作 170
9.4.2 思维定式 171
9.4.3 正则表达式可以匹配各种文本 172
9.4.4 滥用正则表达式 173

第三部分

第 10 章 .NET 176
10.1 预备知识 176
10.2 正则功能详解 177
10.2.1 列表 177
10.2.2 字符组 178
10.2.3 Unicode 属性 178
10.2.4 字符组简记法 179
10.2.5 单词边界 179
10.2.6 行起始/结束位置 180
10.2.7 环视 181
10.2.8 匹配模式 181
10.2.9 捕获分组的引用 182

10.3 正则 API 简介 183
10.3.1 Regex 183
10.3.2 Match 187
10.4 常用操作示例 188
10.4.1 验证 188
10.4.2 提取 189
10.4.3 替换 189
10.4.4 切分 190

第 11 章 Java 191
11.1 预备知识 191
11.2 正则功能详解 192
11.2.1 列表 192
11.2.2 字符组 192
11.2.3 Unicode 属性 194
11.2.4 字符组简记法 194
11.2.5 单词边界 194
11.2.6 行起始/结束位置 195
11.2.7 环视 196
11.2.8 匹配模式 196
11.2.9 纯文本模式 197
11.2.10 捕获分组的引用 197
11.3 正则 API 简介 197
11.3.1 Pattern 198
11.3.2 Matcher 200
11.3.3 String 203
11.4 常用操作示例 204
11.4.1 验证 204
11.4.2 提取 204
11.4.3 替换 205
11.4.4 切分 206
11.5 Java 8 和 Java 9 的新改进 206
11.5.1 Java 8 的新改进 206
11.5.2 Java 9 的新改进 207

第 12 章 JavaScript 208
12.1 预备知识 208
12.2 正则功能详解 209
12.2.1 列表 209
12.2.2 字符组 210
12.2.3 字符组简记法 211
12.2.4 单词边界 211
12.2.5 行起始/结束位置 212
12.2.6 环视 212
12.2.7 匹配模式 213
12.2.8 捕获分组的引用 214
12.3 正则 API 简介 215
12.3.1 RegExp 215
12.3.2 String 218
12.4 常用操作示例 221
12.4.1 验证 221
12.4.2 提取 222
12.4.3 替换 223
12.4.4 切分 223
12.5 关于 ActionScript 223
12.5.1 RegExp 223
12.5.2 匹配规则 224
12.5.3 匹配模式 224
12.5.4 正则 API 224

第 13 章 PHP 225
13.1 预备知识 225
13.2 正则功能详解 227
13.2.1 列表 227
13.2.2 字符组 228
13.2.3 Unicode 属性 229
13.2.4 字符组简记法 229
13.2.5 单词边界 230
13.2.6 行起始/结束位置 230
13.2.7 环视 231
13.2.8 匹配模式 231

13.2.9 纯文本模式 232
13.2.10 捕获分组的引用 232
13.3 正则 API 简介 233
13.3.1 PREG 常量说明 233
13.3.2 preg_quote 235
13.3.3 preg_ grep 235
13.3.4 preg_match 236
13.3.5 preg_match_all 237
13.3.6 preg_last_error 239
13.3.7 preg_replace 239
13.3.8 preg_replace_callback 240
13.3.9 preg_filter 240
13.3.10 preg_split 241
13.3.11 preg_replace_callback_array 242
13.4 常见的正则操作举例 243
13.4.1 验证 243
13.4.2 提取 243
13.4.3 替换 244
13.4.4 切分 244

第 14 章 Python 245
14.1 预备知识 245
14.2 正则功能详解 246
14.2.1 列表 246
14.2.2 字符组 247
14.2.3 Unicode 属性 248
14.2.4 字符组简记法 249
14.2.5 单词边界 250
14.2.6 行起始/结束位置 251
14.2.7 环视 252
14.2.8 匹配模式 252
14.2.9 捕获分组的引用 253
14.2.10 条件匹配 253
14.3 正则 API 简介 254
14.3.1 RegexObject 254
14.3.2 re.compile(regex[, flags]) 255

14.3.3 re.search(pattern, string[, flags]) 256
14.3.4 MatchObject 256
14.3.5 re.match(pattern, string[, flags]) 257
14.3.6 re.findall(pattern, string[, flags]) 258
14.3.7 re.finditer(pattern, string[, flags]) 258
14.3.8 re.split(pattern, string[, maxsplit=0, flags=0]) 259
14.3.9 re.sub(pattern, repl, string[, count, flags]) 259
14.4 常用操作示例 260
14.4.1 验证 260
14.4.2 提取 261
14.4.3 替换 262
14.4.4 切分 262

第 15 章 Ruby 263
15.1 预备知识 263
15.2 正则功能详解 264
15.2.1 列表 264
15.2.2 字符组 264
15.2.3 Unicode 属性 265
15.2.4 字符组简记法 266
15.2.5 单词边界 266
15.2.6 行起始/结束位置 267
15.2.7 环视 268
15.2.8 匹配模式 268
15.2.9 捕获分组的引用 269
15.3 正则 API 简介 269
15.3.1 Regexp 269
15.3.2 Regexp.match(text) 271
15.3.3 Regexp.quote(text)和 Regexp.escape(text) 272
15.3.4 String.index(Regexp) 273
15.3.5 String.scan(Regexp) 273
15.3.6 String.slice(Regexp) 274
15.3.7 String.split(Regexp) 274
15.3.8 String.sub(Regexp, Str) 275
15.3.9 String.gsub(Regexp, String) 276
15.4 常用操作示例 276

15.4.1 验证 276
15.4.2 提取 277
15.4.3 替换 277
15.4.4 切分 277
15.5 Ruby 1.9 的新变化 278

第 16 章 Objective-C 280

16.1 预备知识 280
16.2 正则功能详解 282
16.2.1 列表 282
16.2.2 字符组 283
16.2.3 Unicode 属性 284
16.2.4 字符组简记法 284
16.2.5 单词边界 285
16.2.6 行起始/结束位置 286
16.2.7 环视 287
16.2.8 匹配模式 287
16.2.9 纯文本模式 288
16.2.10 捕获分组的引用 289
16.2.11 命名分组 290
16.3 正则 API 简介 291
16.3.1 predicateWithFormat 291
16.3.2 rangeOfString 292
16.3.3 regularExpressionWithPattern 292
16.3.4 initWithPattern 292
16.3.5 pattern 293
16.3.6 numberOfCaptureGroups 293
16.3.7 numberOfMatchesInString 293
16.3.8 stringByReplacingMatchesInString 294
16.3.9 replacingMatchesInString 294
16.3.10 escapedPatternForString 294
16.3.11 escapedTemplateForString 295
16.4 常用操作示例 295
16.4.1 验证 295
16.4.2 提取 295

16.4.3 替换 297
16.4.4 切分 298

第 17 章 Golang 299
17.1 预备知识 299
17.2 正则功能详解 301
17.2.1 列表 301
17.2.2 字符组 301
17.2.3 Unicode 属性 302
17.2.4 字符组简记法 303
17.2.5 单词边界 303
17.2.6 行起始/结束位置 303
17.2.7 环视 304
17.2.8 匹配模式 304
17.2.9 纯文本模式 305
17.2.10 捕获分组的引用 305
17.2.11 命名分组 306
17.3 正则 API 简介 307
17.3.1 Compile 和 MustCompile 307
17.3.2 MatchString 308
17.3.3 FindString 308
17.3.4 FindAllString 309
17.3.5 FindStringIndex 309
17.3.6 FindAllStringIndex 309
17.3.7 FindStringSubmatch 309
17.3.8 FindAllStringSubmatch 310
17.3.9 SubexpNames 310
17.3.10 Split 311
17.3.11 ReplaceAllString 311
17.3.12 ReplaceAllLiteralString 312
17.4 常用操作示例 312
17.4.1 验证 312
17.4.2 提取 312
17.4.3 替换 313
17.4.4 切分 313

第 18 章 Linux/UNIX 314
18.1 POSIX 314
18.1.1 POSIX 规范 314
18.1.2 POSIX 字符组 316
18.2 vi 317
18.2.1 字符组及简记法 317
18.2.2 量词 318
18.2.3 多选结构和捕获分组 319
18.2.4 环视 319
18.2.5 锚点和单词边界 319
18.2.6 替换操作的特殊字符 320
18.2.7 replacement 中的特殊变量 322
18.2.8 补充 322
18.3 grep 323
18.3.1 基本用法 323
18.3.2 字符组 324
18.3.3 锚点和单词边界 324
18.3.4 量词 324
18.3.5 多选结构和捕获分组 325
18.3.6 options 325
18.3.7 egrep 和 fgrep 326
18.3.8 补充 327
18.4 awk 327
18.4.1 基本用法 327
18.4.2 字符组及简记法 328
18.4.3 锚点和单词边界 329
18.4.4 量词 329
18.4.5 多选结构 330
18.4.6 补充 330
18.5 sed 330
18.5.1 基本用法 330
18.5.2 字符组及简记法 331
18.5.3 锚点和单词边界 331
18.5.4 量词 332
18.5.5 多选结构和捕获分组 332

18.5.6 options.....333
18.5.7 补充.....333
18.6 总结.....334
附录 A 常用语言中正则特性一览.....337
附录 B 常用的正则表达式.....340
附录 C 常用的正则表达式工具及资源.....356
正则表达式术语中英文对照表.....363

第一部分

第 1 章　字符组

1.1　普通字符组

字符组（Character Class）[1]是正则表达式最基本的结构之一，要理解正则表达式的“灵活”，认识它是第一步。

顾名思义，字符组就是“一组”字符，在正则表达式中，它表示“在同一个位置可能出现的各种字符”，其写法是在一对方括号`[`和`]`之间列出所有可能出现的字符，简单的字符组包括`[ab]`、`[314]`、`[#.?]`等。在解决一些常见问题时，使用字符组可以大大简化操作，下面举“匹配数字字符”的例子来说明。

字符可以分为很多类，比如数字、字母、标点等。有时候要求 “只出现一个数字字符”，换句话说，这个位置上的字符只能是 0、1、2、…、8、9 这 10 个字符之一。要进行这种判断，通常的思路是：用 10 个条件分别判断字符是否等于这 10 个字符，对 10 个结果取“或”，只要其中一个条件成立，就返回 True，表示这是一个数字字符，其伪代码如例 1-1 所示。

例 1-1　判断数字字符的伪代码

```
charStr == "0" || charStr == "1" … || charStr == "9"
```

注：因为正则表达式处理的都是“字符串”（String）而不是“字符”，所以这里假设变量 charStr（虽然它只包含一个字符）也是字符串类型，使用了双引号，但是在有些语言中字符串也用单引号表示。

这种解法的问题在于太烦琐——如果要判断一个小写英文字母，就要用||连接 26 个判断；如果还要兼容大写字母，则要连接 52 个判断，代码长到几乎无法阅读。相反，用字符组解决起来却异常简单，具体思路是：列出可能出现的所有字符（在这个例子里就是 10 个数字字符），只要出现了其中任何一个，就返回`True`。例 1-2 给出了使用字符组判断的例子，程序语言使用 Python。

---

[1] 在有的资料中，写作 Character Set，所以也有人将其翻译为“字符类”或者“字符集”。不过在计算机术语中，“类”是和“对象”相关的，“字符集”更适合用来翻译 Character Set（比如 GBK、UCS 之类），所以本书中没有采用这两个名字。

例 1-2 用正则表达式判断数字字符

```
re.search("[0123456789]", charStr) != None
```

`re.search()`是 Python 提供的正则表达式操作函数，表示“进行正则表达式匹配”；*`charStr`* 仍然是需要判断的字符串，而`[0123456789]`则是以字符串形式给出的正则表达式，它是一个字符组，表示“这里可以是 0、1、2、…、8、9 中的任意一个字符。只要 *`charStr`* 与其中任何一个字符相同（或者说“*`charStr`* 可以由`[0123456789]`匹配”），就会得到一个 `MatchObject` 对象（这个对象暂时不必关心，在第 21 页会详细讲解）；否则，返回 `None`。所以判断结果是否为 `None`，就可以判断 *`charStr`* 是否是数字字符。

当今流行的编程语言大多支持正则表达式，上面的例子在各种语言中的写法大抵相同，唯一的区别在于如何调用正则表达式的功能，所以用法其实大同小异。例 1-3 列出了常见语言中的表示，如果你现在就希望知道语言的细节，可以参考本书第三部分的具体章节。

例 1-3 用正则表达式判断数字字符在各种语言中的应用 [1]

.NET（C#）

```
//能匹配则返回 true，否则返回 false
Regex.IsMatch(charStr, "[0123456789]");
```

Java

```
//能匹配则返回 true，否则返回 false
charStr.matches("[0123456789]");
```

JavaScript

```
//能匹配则返回 true，否则返回 false
/[0123456789]/.test(charStr);
```

PHP

```
//能匹配则返回 1，否则返回 0
preg_match("/[0123456789]/", charStr);
```

Python

```
#能匹配则返回 RegexObject，否则返回 None
re.search("[0123456789]", charStr)
```

Ruby

```
#能匹配则返回 0，否则返回 nil
charStr =~ /[0123456789]/
```

[1] 传统上，Perl 是正则表达式处理最方便的编程语言，考虑到今天使用 Perl 的人数，以及 Perl 程序员一般都熟练掌握正则表达式的现实，本书没有给出 Perl 语言的例子。

可以看到，不同语言使用正则表达式的方法也不相同。如果仔细观察会发现，Java、.NET、Python、PHP 中的正则表达式，都要以字符串形式给出，两端都有双引号"；而 Ruby 和 JavaScript 中的正则表达式则不必如此，只在首尾有两个斜线字符/，这也是不同语言中使用正则表达式的不同之处。不过，这个问题现在不需要太关心，因为本书中大部分例子以 Python 程序来讲解，下面讲解关于 Python 的基础知识，其他语言的细节留到后文会详细介绍。

1.2　关于 Python 的基础知识

本书选择使用 Python 语言来演示实际的匹配结果，因为它能在多种操作系统中运行，安装也很方便；另一方面，Python 是解释型语言，输入代码就能看到结果，方便动手实践。考虑到不是所有人都熟悉 Python，这里专门用一节的篇幅来介绍。

如果你的机器上没有安装Python，可以从http://python.org/download/下载，目前Python有 2 和 3 两个版本，本书的例子以 2 版本为准。[1] 请选择自己平台对应的程序下载并安装（目前Mac OS、Linux的各种发行版一般带有Python，具体可以在命令行下输入python，看是否启动对应的程序）。

然后可以启动 Python，在 Mac OS 和 Linux 下输入 python，会显示出 Python 提示符，进入交互模式，如图 1-1 所示（Linux 下的提示符与 Mac OS 下的差不多，所以此处不列出）；而在 Windows 下，需要在“开始”菜单的“程序”中，选择 Python 目录下的 Python(command line)，如图 1-2 所示。

```
Python 2.7.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1–1　Mac OS 下的 Python 提示符

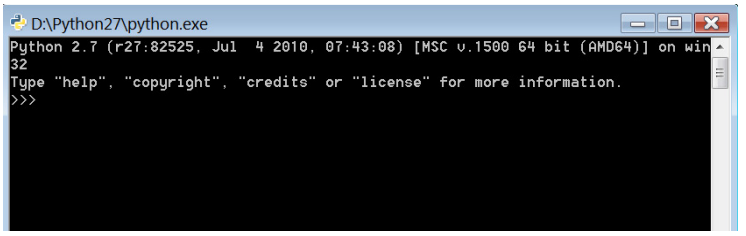


图 1–2　Windows 下的 Python 提示符

Python 中常用的关于正则表达式的函数是 `re.search()`，使用它必须首先导入正则表达式对

[1] 本书初版写作时，2.x 最新的版本为 2.7.1，至修订时最新版本为 2.7.14，正则语法没有明显变化，所以示例仍然以初版为准。Python 3 虽然已经正式发行，但考虑到 2.x 的已有市场份额，以及 3.x 的正则语法变化不大，所以讲解时仍然采用 2.x 版本。关于 2.x 和 3.x 的差别，在 Python 一章有详细介绍。

应的包（package），也就是输入下面的代码。

```
#导入正则表达式对应的包
import re
```

通常的用法是提供两个参数：re.search(*pattern, string*)，其中 *pattern* 是字符串形式提供的正则表达式，*string* 是需要匹配的字符串；如果能匹配，则返回一个 MatchObject（详细介绍请参考第 254 页，暂时可以不必关心），这时提示符会显示类似<_sre.SRE_Match object at 0x0000000001D8E578>之类的结果；如果不能匹配，结果是 None（这是 Python 中的一个特殊值，类似其他某些语言中的 Null），不会有任何显示。图 1-3 演示了运行 Python 语句的结果。

```
>>> import re
>>> re.search("[0123456789]", "4")
<_sre.SRE_Match object at 0x0000000001D8E578>
>>> re.search("[0123456789]", "a")
>>>
```

图 1–3 观察 re.search()匹配的返回值

注：>>>是等待输入的提示符，以>>>开头的行，之后文本是用户输入的语句；其他行是系统生成的，比如打印出语句的结果（在交互模式下，匹配结果会自动输出，便于观察；真正程序运行时不会如此）。

为讲解清楚、形象、方便，本书中的程序部分需要做两点修改。

第一，因为暂时还不需要关心匹配结果的细节，只关心有没有结果，所以在 re.search()之后添加判断返回值是否为 None，如果为 True，则表示匹配成功，否则返回 False 表示匹配失败。为节省版面，尽可能用注释表示这类匹配结果，如# => True 或者 # => False，附在语句之后。

第二，目前我们关心的是整个字符串是否能由正则表达式匹配。但是，在默认情况下 re.search(*pattern, string*)只判断 *string* 的某个子串能否由 *pattern* 匹配，即便 *pattern* 只能匹配 *string* 的一部分，也不会返回 None。为了测试整个 *string* 能否由 *pattern* 匹配，在 *pattern* 两端加上^和$。^和$是正则表达式中的特殊字符，它们并不匹配任何字符，只是表示“定位到字符串的起始位置”和“定位到字符串的结束位置”（原理如图 1-4 所示，如果你现在就希望详细了解这两个特殊字符，可以参考第 62 页），这样就保证，只有在整个 *string* 都可以由 *pattern* 匹配时，才算匹配成功，不返回 None，如例 1-4 所示。

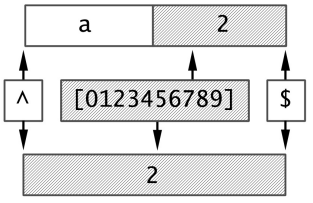


图 1–4 ^[0123456789]$的匹配

例 1-4　使用^和$测试 string 由 pattern 完整匹配

```
# 只要字符串中包含数字字符，就可以匹配
re.search("[0123456789]", "2") != None            #  => True
re.search("^[0123456789]$", "12") != None         #  => False
re.search("[0123456789]", "a2") != None           #  => True

# 整个字符串就是一个数字字符，才可以匹配
re.search("[0123456789]", "2") != None            #  => True
re.search("^[0123456789]$", "12") != None         #  => False
re.search("^[0123456789]$", "a2") != None         #  => False
```

1.3　普通字符组（续）

介绍完关于 Python 的基础知识，继续讲解字符组。字符组中的字符排列顺序**并不影响**字符组的功能，出现重复字符也不会影响，所以`[0123456789]`完全等价于 `[9876543210]`、`[1029384756]`、`[9988876543210]`。

不过，代码总是要容易编写、方便阅读，正则表达式也是一样的，所以一般并不推荐在字符组中出现重复字符。而且，还应该让字符组中的字符排列更符合认知习惯，比如`[0123456789]`就好过`[0192837465]`。为此，正则表达式提供了**-范围表示法**（range），它更直观，能进一步简化字符组。

所谓“-范围表示法”，就是用`[x-y]`的形式表示 x 到 y 整个范围内的字符，省去一一列出的麻烦，这样`[0123456789]`就可以表示为`[0-9]`。如果你觉得这么做看起来意义不大，那么`[a-z]`确实比`[abcdefghijklmnopqrstuvwxyz]`简单太多了。

你可能会问，“-范围表示法”的范围是如何确定的？为什么要写作`[0-9]`，而不写作`[9-0]`？

要回答这个问题，必须了解范围表示法的实质。在字符组中，`-`表示的范围，一般是根据字符对应的**码值**（Code Point，也就是字符在对应编码表中的编码的数值）来确定的，码值小的字符在前，码值大的字符在后。在 ASCII 编码中（包括各种兼容 ASCII 的编码中），字符 `0` 的码值是 48（十进制），字符 `9` 的码值是 57（十进制），所以`[0-9]`等价于`[0123456789]`；而`[9-0]`则是错误的范围，因为 `9` 的码值大于 `0`，所以会报错。程序代码见例 1-5。

例 1-5　[0-9]是合法的，[9-0]会报错

```
re.search("^[0-9]$", "2") != None        #  => True

re.search("^[9-0]$", "2") != None
Traceback (most recent call last):
error: bad character range
```

如果知道 0 ～ 9 的码值是 48 ～ 57，a ～ z 的码值是 97 ～ 122，A ～ Z 的码值是 65 ～ 90，能不能用[0-z]统一表示数字字符、小写字母、大写字母呢？

答案是：勉强可以，但不推荐这么做。根据惯例，字符组的范围表示法用来表示一类字符（数字字符是一类，字母字符也是一类），所以虽然[0-9]、[a-z]都是很好理解的，但[0-z]却很难理解，不熟悉 ASCII 编码表的人甚至不知道这个字符组还能匹配大写字母，更何况，在码值 48 到 122 之间，除去数字字符（码值 48 ～ 57）、小写字母（码值 97 ～ 122）、大写字母（码值 65 ～ 90），还有不少标点符号（参见表 1-1），从字符组[0-z]中却很难看出来，使用时就容易引起误会，例 1-6 所示的程序就很可能让人感觉莫名其妙。

表 1–1 ASCII 编码表（片段）

码值	字符	码值	字符	码值	字符	码值	字符	码值	字符
48	0	63	?	78	N	93	]	108	l
49	1	64	@	79	O	94	^	109	m
50	2	65	A	80	P	95	_	110	n
51	3	66	B	81	Q	96	`	111	o
52	4	67	C	82	R	97	a	112	p
53	5	68	D	83	S	98	b	113	q
54	6	69	E	84	T	99	c	114	r
55	7	70	F	85	U	100	d	115	s
56	8	71	G	86	V	101	e	116	t
57	9	72	H	87	W	102	f	117	u
58	:	73	I	88	X	103	g	118	v
59	;	74	J	89	Y	104	h	119	w
60	<	75	K	90	Z	105	i	120	x
61	=	76	L	91	[	106	j	121	y
62	>	77	M	92	\	107	k	122	z

例 1–6 [0–z]的奇怪匹配

```
re.search("^[0-z]$", "A") != None          #  => True
re.search("^[0-z]$", ":") != None          #  => True
```

在字符组中可以同时并列多个“-范围表示法”，字符组[0-9a-zA-Z]可以匹配数字、大写字母或小写字母；字符组 [0-9a-fA-F]可以匹配数字，大、小写形式的 a ～ f，它可以用来验证十六进制字符，代码见例 1-7。

例 1–7　[0–9a–fA–F]准确判断十六进制字符

```
re.search("^[0-9a-fA-F]$", "0") != None    #  => True
re.search("^[0-9a-fA-F]$", "c") != None    #  => True
re.search("^[0-9a-fA-F]$", "i") != None    #  => False
re.search("^[0-9a-fA-F]$", "C") != None    #  => True
re.search("^[0-9a-fA-F]$", "G") != None    #  => False
```

在不少语言中，还可以用转义序列\x*hex* 来表示一个字符，其中\x 是固定前缀，表示转义序列的开头，*num* 是字符对应的码值（Code Point，详见第 127 页，下文用☞127 表示），是一个两位的十六进制数值。比如字符 A 的码值是 41（十进制则为 65），所以也可以用\x41 表示。

字符组中有时会出现这种表示法，它可以表现一些难以输入或者难以显示的字符，比如\x7F；也可以用来方便地表示某个范围，比如所有 ASCII 字符对应的字符组就是`[\x00-\x7F]`，代码见例 1-8。这种表示法很重要，在第 126 页还会讲到它，依靠这种表示法可以很方便地匹配所有的中文字符。

例 1–8　[\x00–\x7F]准确判断 ASCII 字符

```
re.search("^[\x00-\x7F]$", "c") != None    #  => True
re.search("^[\x00-\x7F]$", "I") != None    #  => True
re.search("^[\x00-\x7F]$", "0") != None    #  => True
re.search("^[\x00-\x7F]$", "<") != None    #  => True
```

1.4　元字符与转义

在上面的例子里，字符组中的横线`-`并不能匹配横线字符，而是用来表示范围，这类字符叫作**元字符**（meta-character）。字符组的开方括号`[`、闭方括号`]`和之前出现的`^`、`$`都算元字符。在匹配中，它们有着特殊的意义。但是，有时候并不需要表示这些特殊意义，只需要表示普通字符（比如“我就想表示横线字符-”），此时就必须做特殊处理。

先来看字符组中的`-`，如果它紧邻着字符组中的开方括号`[`，那么它就是普通字符，其他情况下都是元字符；而对于其他元字符，取消特殊含义的做法都是转义，也就是在正则表达式中的元字符前加上反斜线字符\。

如果要在字符组内部使用横线`-`，最好的办法是将它排列在字符组的开头。`[-09]`就是包含三个字符`-`、`0`、`9` 的字符组；`[0-9]`是包含 `0`~`9` 这 10 个字符的字符组，`[-0-9]`则是由“-范围表示法”`0-9`和横线`-`共同组成的字符组，它可以匹配 11 个字符，例 1-9 说明了使用横线`-`的各种情况。

例 1-9 –出现在不同位置，含义不同

```
#作为普通字符
re.search("^[-09]$", "3") != None        #  => False
re.search("^[-09]$", "-") != None        #  => True
#作为元字符
re.search("^[0-9]$", "3") != None        #  => True
re.search("^[0-9]$", "-") != None        #  => False
#转义之后作为普通字符
re.search("^[0\\-9]$", "3") != None      #  => False
re.search("^[0\\-9]$", "-") != None      #  => True
```

仔细观察会发现，在正文里说“在正则表达式中的元字符之前加上反斜线字符\”，而在代码里写的却不是[0\-9]，而是[0\\-9]。这并不是输入错误。

因为在这段程序里，正则表达式是以字符串（String）的方式[1]提供的，而字符串本身也有关于转义的规定（你或许记得，在字符串中有\n、\t之类的转义序列）。上面说的“正则表达式”，其实是经过“字符串转义处理”之后的字符串的值，正则表达式[0\-9]包含 6 个字符：[、0、\、-、9、]，在表达式中只需要使用这 6 个字符即可，但是在源代码里，必须使用 7 个字符：\需要转义成\\，因为处理字符串时，反斜线和它之后的字符会被认为是转义序列（Escape Sequence），比如\n、\t都是合法的转义序列，然而\-不是。

这个问题确实有点麻烦。正则表达式是用来处理字符串的，但它又不完全等于字符串，正则表达式中的每个反斜线字符\，在字符串中（也就是正则表达式之外）还必须转义为\\。所以之前所说的是“正则表达式[0\-9]”，程序里写的却是[0\\-9]，这确实有点麻烦，但需要注意。

不过，Python 提供了**原生字符串**（Raw String），它非常适合于正则表达式：正则表达式是怎样的，原生字符串就是怎样的，完全不需要考虑正则表达式之外的转义（只有双引号字符是例外，原生字符串内的双引号字符必须转义写成\"）。原生字符串的形式是 r"*string*"，也就是在普通字符串之前添加 r，示例代码如例 1-10。

例 1-10 原生字符串的使用

```
#原生字符串和字符串的等价
r"^[0\-9]$" == "^[0\\-9]$"               #  => True
#原生字符串的转义要简单许多
re.search(r"^[0\-9]$", "3") != None      #   => False
re.search(r"^[0\-9]$", "-") != None      #   => True
```

原生字符串清晰易懂，省去了转义的麻烦，所以从现在开始，本书中的 Python 示范代码都

[1] 具体来说，在 Java、PHP、Python、.NET 等语言中，正则表达式都是以字符串的形式给出的，在 Ruby 和 JavaScript 中则不是这样。详细的说明，请参考第 96 页。

使用原生字符串来表示正则表达式。另外，.NET 和 Ruby 中也有原生字符串，也有一些语言并没有提供原生字符串（比如 Java），所以在第 6 章（☞95）会专门讲解转义问题。不过，现在只需要知道 Python 示范代码中使用了原生字符串即可。

继续看转义，如果希望在字符组中列出闭方括号`]`，比如`[012]345]`，就必须在它之前使用反斜线转义，写成`[012\]345]`；否则，结果就如例 1-11 所示，正则表达式将`]`与最近的`[`匹配，这个表达式就成了“字符组`[012]`加上 4 个字符 `345]`”，它能匹配的是字符串 `0345]`或 `1345]`或 `2345]`，却不能匹配`]`。

例 1–11　]出现在不同位置，含义不同

```
#未转义的]
re.search(r"^[012]345]$", "2345") != None          #  => True
re.search(r"^[012]345]$", "5") != None             #  => False
re.search(r"^[012]345]$", "]") != None             #  => False
#转义的]
re.search(r"^[012\]345]$", "2345") != None         #  => False
re.search(r"^[012\]345]$", "5") != None            #  => True
re.search(r"^[012\]345]$", "]") != None            #  => True
```

除去字符组内部的`-`，其他元字符的转义都必须在字符之前添加反斜线，`[`的转义也是如此。如果只希望匹配字符串`[012]`，直接使用正则表达式`[012]`是不行的，因为这会被识别为一整个字符组，它只能匹配 `0`、`1`、`2` 这三个字符中的任意一个；所以必须转义，把正则表达式写作`\[012]`，请注意，只有开方括号`[`需要转义，闭方括号`]`不需要转义，如例 1-12 所示。

例 1–12　取消其他元字符的特殊含义

```
re.search(r"^[012]345]$", "3") != None         #  => False
re.search(r"^[012\\]345]$", "3") != None       #  => True
re.search(r"^[012]$", "[012]") != None         #  => False
re.search(r"^\[012]$", "[012]") != None        #  => True
```

1.5　排除型字符组

在方括号`[…]`中列出希望匹配的所有字符，这种字符组可以叫作“普通字符组”，它的确非常方便。不过，也有些问题是普通字符组不能解决的。

给定一个由两个字符构成的字符串*`str`*，要判断这两个字符是否都是数字字符，可以用`[0-9][0-9]`来匹配。但是，如果要求判断的是这样的字符串——第一个字符不是数字字符，第二个字符才是数字字符（比如`A8`、`x6`）[1]——应当如何处理？数字字符的匹配很好处理，用`[0-9]`

[1] 一般来说，计算机中的偏移值都是从 0 开始的。此处考虑到叙述自然，使用了“第一个字符”和“第二个字符”的说法，其中“第一个字符”指最左端，也就是偏移值为 0 的字符；“第二个字符”指紧跟在它右侧，也就是偏移值为 1 的字符。

即可；“不是数字”则很难办——不是数字的字符太多了，全部列出几乎不可能，这时就应当使用排除型字符组。

排除型字符组（Negated Character Class）非常类似普通字符组`[…]`，只是在开方括号[之后紧跟一个脱字符^，写作`[^…]`，表示“在当前位置，匹配一个没有列出的字符”。所以`[^0-9]`就表示“0 ~ 9之外的字符”，也就是“非数字字符”。那么，`[^0-9][0-9]`就可以解决问题了，如例 1-13 所示。

例 1–13 使用排除型字符组

```
re.search(r"^[^0-9][0-9]$", "A8") != None          #  => True
re.search(r"^[^0-9][0-9]$", "x6") != None          #  => True
```

排除型字符组看起来很简单，不过新手常常会犯一个错误，就是把“在当前位置，**匹配一个**没有列出的字符”理解成“在当前位置**不要匹配**列出的字符”，两者其实是不同的，后者暗示“这里**不出现任何字符也可以**”。例 1-14 很清楚地说明：**排除型字符组必须匹配一个字符**，这点一定要记住。

例 1–14 排除型字符组必须匹配一个字符

```
re.search(r"^[^0-9][0-9]$", "8") != None       #  => False
re.search(r"^[^0-9][0-9]$", "A8") != None      #  => True
```

除了开方括号[之后的^，排除型字符组的用法与普通字符组几乎完全相同，唯一需要改动的是：在排除型字符组中，如果需要表示横线字符-（而不是用于“-范围表示法”），那么-应该紧跟在^之后；而在普通字符组中，作为普通字符的横线-应该紧跟在开方括号之后，如例 1-15 所示。

例 1–15 在排除型字符组中，紧跟在^之后的–不是元字符

```
#匹配一个-、0、9之外的字符
re.search(r"^[^-09]$", "-") != None            #  => False
re.search(r"^[^-09]$", "8") != None            #  => True

#匹配一个0~9之外的字符
re.search(r"^[^0-9]$", "-") != None            #  => True
re.search(r"^[^0-9]$", "8") != None            #  => False
```

在排除型字符组中，^是一个元字符，但只有它紧跟在[之后时才是元字符，如果想表示“这个字符组中可以出现^字符”，不要让它紧挨着[即可，否则就要转义。例 1-16 给出了三个正则表达式，后两个表达式实质是一样的，但第三种写法很麻烦，理解起来也麻烦，不推荐使用。

例 1-16　排除型字符组的转义

```
#匹配一个 0、1、2 之外的字符
re.search(r"^[^012]$", "^") != None          #  => True
#匹配 4 个字符之一：0、^、1、2
re.search(r"^[0^12]$", "^") != None          #  => True
#^紧跟在[之后，但经过转义变为普通字符，等于上一个表达式，不推荐
re.search(r"^[\^012]$", "^") != None         #  => True
```

1.6　字符组简记法

用`[0-9]`、`[a-z]`等字符组，可以很方便地表示数字字符和小写字母字符。对于这类常用的字符组，正则表达式提供了更简单的记法，这就是**字符组简记法**（shorthands）。

常见的字符组简记法有`\d`、`\w`、`\s`。从表面上看，它们与`[…]`完全没联系，但效果其实是等价的。其中`\d`等价于`[0-9]`，其中的 d 代表“数字（digit）”；`\w`等价于`[0-9a-zA-Z_]`，其中的 w 代表“单词字符（word）”；`\s`等价于`[ \t\r\n\v\f]`（第一个字符是空格），s 表示“空白字符（space）”。例 1-17 说明了这几个字符组简记法的典型匹配。

例 1-17　字符组简记法\d、\w、\s

```
#注意，如果没有原生字符串，\d 就必须写作\\d
re.search(r"^\d$", "8") != None       #  => True
re.search(r"^\d$", "a") != None       #  => False

re.search(r"^\w$", "8") != None      #  => True
re.search(r"^\w$", "a") != None      #  => True
re.search(r"^\w$", "_") != None      #  => True

re.search(r"^\s$", " ") != None      #  => True
re.search(r"^\s$", "\t") != None     #  => True
re.search(r"^\s$", "\n") != None     #  => True
```

一般印象中，单词字符似乎只包含大小写字母，但是字符组简记法中的“单词字符”不只有大小写单词，还包括数字字符和下画线_，其中的下画线_尤其值得注意：在进行数据验证时，有可能只允许输入“数字和字母”，有人会偷懒用`\w`验证，而忽略了`\w`能匹配下画线，所以这种匹配并不严格，`[0-9a-zA-Z]`才是准确的选择。

“空白字符”并不难定义，它可以是空格字符、制表符`\t`、回车符`\r`、换行符`\n`等各种“空白”字符，只是不方便展现（因为显示和印刷出来都是空白）。不过这也提醒我们注意，匹配时看到的“空白”可能不是空格字符，因此，`\s`才是准确的选择。

字符组简记法可以单独出现，也**可以使用在字符组中**[1]，比如`[0-9a-zA-Z]`也可以写作

[1] 只有 Java 8 引入的字符组简记法是`\R`例外，但它并不是一个通用的字符组简记法。

[\da-zA-Z]，所以匹配十六进制字符的字符组可以写成[\da-fA-F]。字符组简记法也可以用在排除型字符组中，比如[^0-9]就可以写成[^\d]，[^0-9a-zA-Z_]就可以写成[^\w]，代码如例 1-18。

例 1–18 字符组简记法与普通字符组混用

```
#用在普通字符组内部
re.search(r"^[\da-zA-Z]$", "8") != None    #  => True
re.search(r"^[\da-zA-Z]$", "a") != None    #  => True
re.search(r"^[\da-zA-Z]$", "C") != None    #  => True
#用在排除型字符组内部
re.search(r"^[^\w]$", "8") != None         #  => False
re.search(r"^[^\w]$", "_") != None         #  => False
re.search(r"^[^\w]$", ",") != None         #  => True
```

相对于\d、\w和\s这三个普通字符组简记法，正则表达式也提供了对应排除型字符组的简记法：\D、\W和\S——字母完全一样，只是改为大写。这些简记法能匹配的字符是互补的：\s能匹配的字符，\S一定不能匹配；\w能匹配的字符，\W一定不能匹配；\d能匹配的字符，\D一定不能匹配。例 1-19 示范了这几个字符组简记法的应用。

例 1–19 \D、\W、\S 的使用

```
#\d 和\D
re.search(r"^\d$", "8") != None       #  => True
re.search(r"^\d$", "a") != None       #  => False
re.search(r"^\D$", "8") != None       #  => False
re.search(r"^\D$", "a") != None       #  => True
#\w 和\W
re.search(r"^\w$", "c") != None       #  => True
re.search(r"^\w$", "!") != None       #  => False
re.search(r"^\W$", "c") != None       #  => False
re.search(r"^\W$", "!") != None       #  => True
#\s 和\S
re.search(r"^\s$", "\t") != None      #  => True
re.search(r"^\s$", "0") != None       #  => False
re.search(r"^\S$", "\t") != None      #  => False
re.search(r"^\S$", "0") != None       #  => True
```

妥善利用这种互补的属性，可以得到一些非常巧妙的效果，最简单的应用就是字符组[\s\S]。初看起来，在同一个字符组中并列两个互补的简记法，这种做法有点奇怪，不过仔细想想就会明白，\s和\S组合在一起，匹配的就是“所有的字符”（或者叫“任意字符”）。许多语言中的正则表达式并没有直接提供“任意字符”的表示法，所以[\s\S]、[\w\W]、[\d\D]虽然看起来有点古怪，但确实可以匹配任意字符。[1]

[1] 许多关于正则表达式的文档说：点号.能匹配“任意字符”。但在默认情况下，点号其实不能匹配换行符，具体请参考第 86 页。

关于字符组简记法，最后需要补充三点：第一，如果字符组中出现了字符组简记法，最好不要出现单独的-，否则可能引起错误，比如`[\d-a]`就很让人迷惑，在有些语言中，-会被作为普通字符，而在有些语言中，这样写会报错；第二，以上说的`\d`、`\w`、`\s`的匹配规则，都是针对 ASCII 编码而言的，也叫 **ASCII 匹配规则**。但是，目前一些语言中的正则表达式已经支持了 Unicode 字符，那么数字字符、单词字符、空白字符的范围，已经不仅限于 ASCII 编码中的字符。关于这个问题，具体情况在后文有详细的介绍，如果你现在就想知道，可以翻到第 120 页；第三，不同的语言可能有一些专属的独特的字符组简记法，比如 Java 8 就提供了`\h`、`\v`两种，前者匹配“任何水平方向的空白字符”，后者匹配“任何垂直方向的空白字符”。在某些具体场景下，它们确实很方便，只是不熟悉的读者阅读起来会有点困难。

1.7　字符组运算

以上介绍了字符组的基本功能，它们在常用的语言中都有提供；还有些语言为字符组提供了更强大的功能，比如 Java、.NET、Objective-C 就提供了字符组运算的功能，可以在字符组内进行集合运算，在某些情况下这种功能非常实用。

如果要匹配所有的元音字母（为讲解简单考虑，暂时只考虑小写字母的情况），可以用`[aeiou]`，但是要匹配所有的辅音字母却没有什么方便的办法，最直接的写法是`[b-df-hj-np-tv-z]`，不但烦琐，而且难理解。换个角度看，从 26 个字母中“减去”元音字母，剩下的就是辅音字母，如果有办法做这个“减法”，就方便多了。

Java 语言中提供了这样的字符组：`[[a-z]&&[^aeiou]]`，虽然初看有点古怪，但仔细看看，也不难理解。`[a-z]`表示 26 个英文字母，`[^aeiou]`表示除元音字母之外的所有字符（还包括大写字母、数字和各种符号），两者取交集，就得到“26 个英文字母中，除去 5 个元音字母，剩下的 21 个辅音字母”。

.NET 中也有这样的功能，只是写法不一样。同样是匹配辅音字母的字符组，.NET 中写作`[a-z-[aeiou]]`，其逻辑是：从`[a-z]`能匹配的 26 个字符中，“减去”`[aeiou]`能匹配的元音字母。相对于 Java，这种逻辑更符合直觉，但写法却有点古怪——不是`[[a-z]-[aeiou]]`，而是`[a-z-[aeiou]]`。例 1-20 集中演示了 Java 和.NET 中的字符组运算。

例 1–20　字符组运算

```
Java
"a".matches("^[[a-z]&&[^aeiou]]$");       //  => False
"b".matches("^[[a-z]&&[^aeiou]]$");       //  => True

.NET
Regex.IsMatch("^[a-z-[aeiou]]$", "a");    //  => False
Regex.IsMatch("^[a-z-[aeiou]]$", "b");    //  => True
```

1.8 POSIX 字符组

前面介绍了常用的字符组，但是在某些文档中，你可能会发现类似`[:digit:]`、`[:lower:]`之类的字符组，看起来不难理解（digit 就是“数字”，lower 就是“小写”），但又很奇怪，它们就是 **POSIX 字符组**（POSIX Character Class）。因为某些语言的文档中出现了这些字符组，为避免困惑，这里有必要做一个简要介绍。如果只使用常用的编程语言，可以忽略文档中的 POSIX 字符组，也可以忽略本节；如果想了解 POSIX 字符组，或者需要在 Linux/UNIX 下的各种工具（sed、awk、grep 等）中使用正则表达式，最好阅读本节。

之前介绍的字符组，都属于 Perl 衍生出来的正则表达式流派（Flavor），这个流派叫作 **PCRE**（Per Compatible Regular Expression）。除此之外，正则表达式还有其他流派，比如 POSIX（Portable Operating System Interface for uniX），它是一系列规范，定义了 UNIX 操作系统应当支持的功能，其中也包括关于正则表达式的规范，`[:digit:]`之类的字符组就是遵循 POSIX 规范的字符组。

常见的`[a-z]`形式的字符组，在 POSIX 规范中仍然获得支持，它的准确名称是 **POSIX 方括号表达式**（POSIX bracket expression），主要用在 UNIX/Linux 系统中。POSIX 方括号表达式与之前所说的字符组最主要的差别在于：在 POSIX 字符组中，反斜线\不是用来转义的。所以 POSIX 方括号表达式`[\d]`只能匹配\和 d 两个字符，而不是`[0-9]`对应的数字字符。

为了解决字符组中特殊意义字符的转义问题，POSIX 方括号表达式规定：如果要在字符组中表达字符]（而不是作为字符组的结束标记），应当让它紧跟在字符组的开方括号之后，所以`[]a]`能匹配的字符就是]或 a；如果要在字符组中标识字符-（而不是“-范围表示法”），就必须将它放在字符组的闭方括号]之前，所以`[a-]`能匹配的字符就是 a 或-。

另一方面，POSIX 规范还定义了 **POSIX 字符组**（POSIX character class），它大致等于之前介绍的字符组简记法，都是使用类似`[:digit:]`、`[:lower:]`之类有明确意义的记号表示某类字符。

表 1-2 简要介绍了 POSIX 字符组，注意表格中与其对应的是 ASCII 字符组，也就是能匹配的 ASCII 字符（ASCII 编码表中码值在 0~127 之间的字符）。因为 POSIX 规范中有一个重要概念：locale（通常翻译为“语言环境”），它是一组与语言和文化相关的设定，包括日期格式、货币币值、字符编码等。POSIX 字符组的意义会根据 locale 的变化而变化，表 1-2 介绍的只是这些 POSIX 字符组在 ASCII 编码中的意义；如果换用其他的 locale（比如使用 Unicode 字符集），它们的意义可能会发生变化，具体请参考第 135 页。

表 1–2 POSIX 字符组

POSIX 字符组	说明	ASCII 字符组	等价的 PCRE 简记法
`[:alnum:]`*	字母字符和数字字符	`[a-zA-Z0-9]`	
`[:alpha:]`	字母	`[a-zA-Z]`	

（续表）

POSIX 字符组	说明	ASCII 字符组	等价的 PCRE 简记法
`[:ASCII:]`	ASCII 字符	`[\x00-\x7F]`	
`[:blank:]`	空格字符和制表符	`[ \t]`	
`[:cntrl:]`	控制字符	`[\x00-\x1F\x7F]`	
`[:digit:]`	数字字符	`[0-9]`	`\d`
`[:graph:]`	空白字符之外的字符	`[\x21-\x7E]`	
`[:lower:]`	小写字母字符	`[a-z]`	
`[:print:]`	类似`[:graph:]`，但包括空白字符	`[\x20-\x7E]`	
`[:punct:]`	标点符号	`[][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]`	
`[:space:]`	空白字符	`[ \t\r\n\v\f]`	`\s`
`[:upper:]`	大写字母字符	`[A-Z]`	
`[:word:]`*	字母字符	`[A-Za-z0-9_]`	`\w`
`[:xdigit:]`	十六进制字符	`[A-Fa-f0-9]`	

注：标记*的字符组简记法并不是 POSIX 规范中的，但使用很多，一般语言中都提供，文档中也会出现。

POSIX 字符组的使用也与 PCRE 字符组简记法的使用有所不同，主要区别在于，PCRE 字符组简记法可以脱离方括号直接出现，而 POSIX 字符组必须出现在方括号内。所以同样是匹配数字字符，PCRE 中可以直接写`\d`，而 POSIX 字符组必须写成`[[:digit:]]`。

在本书介绍的各种语言中，Java、PHP、Ruby、Golang 支持使用 POSIX 字符组。

在 PHP 中可以直接使用 POSIX 字符组，但是 PHP 中的 POSIX 字符组只识别 ASCII 字符，也就是说，任何非 ASCII 字符（比如中文字符）都不能由任何一个 POSIX 字符组匹配。

Ruby 的情况稍微复杂一点。Ruby 1.8 中的 POSIX 字符组只能匹配 ASCII 字符，而且不支持`[:word:]`和`[:ASCII:]`；Ruby 1.9 中的 POSIX 字符组可以匹配 Unicode 字符，而且支持`[:word:]`和`[:ASCII:]`。

Java 中的情况更加复杂。POSIX 字符组`[[:name:]]`必须使用`\p{name}`的形式，其中 *name* 为 POSIX 字符组对应的名字，比如`[:space:]`就应当写作`\p{Space}`，请注意第一个字母要大写，其他 POSIX 字符组都是这样，只有`[:xdigit:]`要写作`\p{XDigit}`。还需要指出的是，Java 中的 POSIX 字符组只能匹配 ASCII 字符。

Golang 的情况与 PHP 类似，POSIX 字符组可以直接使用，但是都只能匹配 ASCII 字符。

第 2 章　量词

2.1　一般形式

根据第 1 章的介绍，可以用字符组`[0-9]`或者`\d`匹配单个数字字符。现在用正则表达式来验证更复杂的字符串，比如中国大陆地区的邮政编码。

粗略来看，邮政编码并没有特殊的规定，只是 6 位数字构成的字符串，比如 201203、100858，所以用正则表达式来表示就是`\d\d\d\d\d\d`，如例 2-1 所示，只有同时满足“长度是 6 个字符”和“每个字符都是数字”这两个条件，匹配才成功（同样，这里不能忽略`^`和`$`）。

例 2-1　匹配邮政编码

```
re.search(r"^\d\d\d\d\d\d$", "100859") != None        #  => True
re.search(r"^\d\d\d\d\d\d$", "201203") != None        #  => True

re.search(r"^\d\d\d\d\d\d$", "20A203") != None        #  => False
re.search(r"^\d\d\d\d\d\d$", "20103") != None         #  => False
re.search(r"^\d\d\d\d\d\d$", "2012036") != None       #  => False
```

虽然这不难理解，但`\d`重复了 6 次，读写都不方便。为此，正则表达式提供了**量词**（quantifier），比如上面匹配邮政编码的表达式，就可以如例 2-2 那样，简写为`\d{6}`，它使用阿拉伯数字，更简洁也更直观。

例 2-2　使用量词简化字符组

```
re.search(r"^\d{6}$", "100859") != None       #  => True
re.search(r"^\d{6}$", "201203") != None       #  => True

re.search(r"^\d{6}$", "20A203") != None       #  => False
re.search(r"^\d{6}$", "20103") != None        #  => False
re.search(r"^\d{6}$", "2012036") != None      #  => False
```

量词还可以表示不确定的长度，其通用形式是`{m,n}`，其中*m*和*n*是两个数字（有些人习惯在代码中的逗号之后添加空格，这样更好看，但是量词中的逗号之后**不能**有空格），它限定之前的

元素[1]能够出现的次数，*m*是下限，*n*是上限（均为闭区间）。比如`\d{4,6}`，表示这个数字字符串的长度最短是 4 个字符（“单个数字字符”至少出现 4 次），最长是 6 个字符。

如果不确定长度的上限，也可以省略，只指定下限，写成`\d{m,}`，比如`\d{4,}`表示“数字字符串的长度必须在 4 个字符以上”。

量词限定的出现次数一般都有明确下限，如果没有，则默认为 0。有一些语言（比如 Ruby）支持`{,n}`的记法，这时候并不是“不确定长度的下限”，而是省略了“下限为 0”的情况，比如`\d{,6}`表示“数字字符串最多可以有 6 个字符”。不过，这种用法并不是在所有语言中都通用的，比如 Java 就不支持这种写法，所以必须写明`{0,n}`。我推荐的做法是：最好使用`{0,n}`的记法，因为它是被广泛支持的。表 2-1 集中说明了这几种形式的量词，例 2-3 展示了它们的使用。

表 2–1　量词的一般形式

量词	说明
{n}	之前的元素必须出现 *n* 次
{m,n}	之前的元素最少出现 *m* 次，最多出现 *n* 次
{m,}	之前的元素最少出现 *m* 次，出现次数无上限
{0,n}	之前的元素可以不出现，也可以出现，最多出现 *n* 次（在某些语言中可以写为{,n}）

例 2–3　表示不确定长度的量词

```
re.search(r"^\d{4,6}$", "123") != None        #  => False
re.search(r"^\d{4,6}$", "1234") != None       #  => True
re.search(r"^\d{4,6}$", "123456") != None     #  => True
re.search(r"^\d{4,6}$", "1234567") != None    #  => False

re.search(r"^\d{4,}$", "123") != None         #  => False
re.search(r"^\d{4,}$", "1234") != None        #  => True
re.search(r"^\d{4,}$", "123456") != None      #  => True

re.search(r"^\d{0,6}$", "12345") != None      #  => True
re.search(r"^\d{0,6}$", "123456") != None     #  => True
re.search(r"^\d{0,6}$", "1234567") != None    #  => False
```

读者可能会好奇，有些量词没有指定上限，那么重复次数真的没有上限吗？重复亿万次也不会有问题吗？这是一个好问题，严谨的程序员当然希望知道答案。虽然普通的文档没有说明，但我们可以查阅相关的源代码来找到答案——通常，这个隐式的上限是 65 536。按照我使用正则表达式的经验，在绝大多数情况下，这个上限是足够的，所以一般可以认为“不存在上限”。

---

[1] 在第 1 章中提到，字符组是正则表达式的基本“结构”之一，而此处提到之前的“元素”，在此做一点解释。在本书中，“结构”一般指的是正则表达式所提供功能的记法。比如字符组就是一种结构，第 3 章要提到的括号也是一种结构；而“元素”指的是具体的正则表达式中的某个部分，比如某个具体表达式中的字符组`[a-z]`，可以算作一个元素，“元素”也叫“子表达式”（sub-expression）。

2.2 常用量词

{m,n}是通用形式的量词，正则表达式还有 3 个常用量词，分别是+、?、*。它们的形态虽然不同于{m,n}，功能却是相同的（也可以把它们理解为“量词简记法”），具体说明见表 2-2。

表 2-2 常用量词

常用量词	{m,n}等价形式	说明
*	{0,}	可能出现，也可能不出现，出现次数没有上限
+	{1,}	至少出现 1 次，出现次数没有上限
?	{0,1}	至多出现 1 次，也可能不出现

在实际应用中，在很多情况下只需要表示这三种意思，所以常用量词的使用频率要高于{m,n}，下面分别说明。

大家都知道，美国英语和英国英语有些词的写法是不一样的，比如 traveler 和 traveller，如果希望“通吃”traveler 和 traveller，就要求第 2 个 l 是“至多出现 1 次，也可能不出现”的，正好使用?量词：travell?er，如例 2-4 所示。

例 2-4 量词?的应用

```
re.search(r"^travell?er$", "traveler") != None     #  => True
re.search(r"^travell?er$", "traveller") != None    #  => True
```

其实这样的情况还有很多，比如favor和favour、color和colour。此外还有很多其他应用场合，比如http和https，虽然是两个概念，但都是协议名，可以用https?匹配；再比如表示价格的字符串，有可能是 100 也有可能是￥100，可以用￥?100匹配。[1]

量词也广泛应用于解析HTML代码。HTML是一种“标签语言”，它包含各种各样的tag（标签），比如<head>、<img>、<table>等，这些tag的名字各异，形式却相同：从<开始，到>结束，在<和>之间有若干字符，“若干”的意思是长度不确定，但不能为 0（<>并不是合法的tag），也不能是>字符。[2] 如果要用一个正则表达式匹配所有的tag，需要用<匹配开头的<，用>匹配结尾的>，用[^>]+匹配中间的“若干字符”，所以整个正则表达式就是<[^>]+>，程序如例 2-5 所示。

---

[1] 实际上，这个问题比较复杂，因为￥并不是一个 ASCII 字符，所以￥?可能会遇到古怪的问题，具体情况请参考第 7 章。

[2] 如果你对 HTML 代码比较了解，可能会有疑问，假如 tag 内部出现>符号，怎么办？这种情况确实存在，比如<input name=txt value=">">。以目前已经讲解的知识还无法解决这个问题，不过第 3 章就会给出它的解法。

例 2–5　量词+的应用

```
re.search(r"^<[^>]+>$", "<bold>") != None          #  => True
re.search(r"^<[^>]+>$", "</table>") != None        #  => True
re.search(r"^<[^>]+>$", "<>") != None              #  => False
```

类似的，也可以使用正则表达式匹配双引号字符串。不同的是，双引号字符串的两个双引号之间可以没有任何字符，""是一个完全合法的字符串。应该使用量词*，于是整个正则表达式就成了"[^"]*"，程序见例 2-6。

例 2–6　量词*的应用

```
re.search(r"^\"[^\"]*\"$", "\"some\"") != None     #  => True
re.search(r"^\"[^\"]*\"$", "\"\"") != None         #  => True
```

注：字符串中表示双引号需要转义写成\"，这并不是正则表达式中的规定，而是为字符串转义考虑。

量词的使用有很多学问，不妨多看几个 tag 匹配的例子：tag 可以粗略分为 open tag 和 close tag，比如<head>就是 open tag，而</html>就是 close tag；另外还有一类标签是 self-closing tag，比如
。现在来看分别匹配这三类 tag 的正则表达式。

open tag 的特点是以<开头，然后是“若干字符”（但不能以/开头），最后是>，所以对应的正则表达式是<[^/][^>]*>；注意，因为[^/]必须匹配一个字符，所以“若干字符”中其他部分必须写成[^>]*，否则它无法匹配名字为单个字符的标签，比如<b>。

close tag 的特点是以<开头，之后是/字符，然后是“若干字符（但不能以/开头）”，最后是>，所以对应的正则表达式是</[^>]+>；

self-closing tag 的特点是以<开头，中间是“若干字符”，最后是/>，所以对应的正则表达式是<[^>]+/>。注意：这里不是<[^>/]+/>，排除型字符组只排除>，而不排除/，因为要确认的只是在结尾的>之前出现/，如果写成<[^>/]+/>，则要求 tag 内部不能出现/，就无法匹配这类的 tag 了。

表 2-3 列出了匹配几类 tag 的表达式。

表 2–3　各类 tag 的匹配

匹配所有 tag 的表达式	tag 分类	匹配分类 tag 的表达式
<[^>]+>	open tag	<[^/][^>]*>
	close tag	</[^>]+>
	self-closing tag	<[^>/]+/>

对比表格中“匹配所有 tag 的表达式”和“匹配分类 tag 的表达式”，可以发现它们的模式是相近的，只是细节上有差异。也就是说，通过变换字符组和量词，可以准确控制正则表达式能匹

配的字符串的范围，达到不同的目的。这其实是使用正则表达式时的一条根本规律：**使用合适的结构（包括字符组和量词），精确表达自己的意图，界定能匹配的文本**。

再仔细观察，你或许会发现，匹配 open tag 的表达式，也可以匹配 self-closing tag：`<[^/][^>]*>`能够匹配`<br/>`，因为`[^>]*`并不排除对`/`的匹配。那么将表达式改为`<[^/][^>]*[^/]>`，就保证匹配的 open tag 不会以`/>`结尾了。

不过这会产生新的问题：`<[^/][^>]*[^/]>`能匹配的 tag，在`<`和`>`之间出现了两个`[^/]`，上一章已经讲过，排除型字符组表示“在当前位置，匹配一个没有列出的字符”，所以 tag 里的字符串必须至少包含两个字符，这样就无法匹配`<u>`了。

仔细想想，真正要表达的意思是，在 tag 内部的字符串不能以`/`开头，也不能以`/`结尾，如果这个字符串只包含一个字符，那么它既是开头，又是结尾，使用两个排除型字符组显然是不合适的，看起来没办法解决了。其实，也只是现有的知识还不足够解决这个问题而已，在第 69 页有这个问题的详细解法。

2.3 数据提取

正则表达式的功能很多，除去之前介绍的验证（字符串能否由正则表达式匹配），还可以从某个字符串中提取出某个字符串能匹配的所有文本。

第 1 章提到，`re.search()`如果匹配成功，返回一个 `MatchObject` 对象。这个对象包含了匹配的信息，比如表达式匹配的结果，可以像例 2-7 那样，通过调用 `MatchObject.`*`group`*`(0)`来获得。这个方法以后会详细介绍，现在只需要了解一点：调用它可以得到表达式匹配的文本。

例 2–7　通过 MatchObject 获得匹配的文本

```
#注意这里使用链式编程
print re.search(r"\d{6}", "ab123456cd").group(0)
123456
print re.search(r"^<[^>]+>$", "<bold>").group(0)
<bold>
```

这里再介绍一个方法：`re.findall(`*`pattern`*`,` *`string`*`)`。其中 *`pattern`* 是正则表达式，*`string`* 是字符串。这个方法会返回一个数组，其中的元素是在 *`string`* 中依次寻找 *`pattern`* 能匹配的文本。

以邮政编码的匹配为例，假设某个字符串中包含两个邮政编码：`zipcode1:201203, zipcode2:100859`，仍然使用之前匹配邮政编码的正则表达式`\d{6}`，调用 `re.`*`findall`*`()`可以将这两个邮政编码提取出来，如例 2-8 所示。注意，这次要去掉表达式首尾的`^`和`$`，因为要使用正则表达式在字符串中寻找匹配，而不是验证整个字符串能否由正则表达式匹配。

例 2–8　使用 re.findall()提取数据

```
print re.findall(r"\d{6}", "zipcode1:201203, zipcode2:100859")
['201203', '100859']

#也可以逐个输出
for zipcode in re.findall(r"\d{6}", "zipcode1:201203, zipcode2:100859"):
    print zipcode
201203
100859
```

借助之前匹配各种 tag 的正则表达式，还可以通过 `re.findall()`将某个 HTML 页面中所有的 tag 提取出来，下面以 Yahoo 首页为例。

首先要读入 http://www.yahoo.com/的 HTML 源代码，在 Python 中先获得 URL 对应页面的源代码，保存到 *`htmlSource`* 变量中，然后针对匹配各类 tag 的正则表达式，分别调用 `re.findall()`，获得各类 tag 的列表（因为这个页面中包含的 tag 太多，每类 tag 只显示前 3 个）。

因为这段程序的输出很多，在交互式界面下不方便操作和观察，建议将这些代码单独保存为一个.py 文件，比如 `findtags.py`，然后输入 `python findtags.py` 运行。如果输入 `python` 没有结果（一般在 Windows 下会出现这种情况），需要准确设定 `PATH` 变量，比如 `d:\Python\python`。之后，就会看到例 2-9 显示的结果。

例 2–9　使用 re.findall()提取 tag

```
#导入需要的 package
import urllib
import re
#读入 HTML 源代码
sock = urllib.urlopen("http://yahoo.org/")
htmlSource = sock.read()
sock.close()
#匹配，输出结果（[0:3]表示取前 3 个）
print "open tags:"
print re.findall(r"<[^/>][^>]*[^/>]>", htmlSource)[0:3]
print "close tags:"
print re.findall(r"</[^>]+>", htmlSource) [0:3]
print "self-closing tags:"
print re.findall(r"<[^>/]+/>", htmlSource) [0:3]

open tags:
['<!DOCTYPE html>', '<html lang="en-US" class="y-fp-bg y-fp-pg-grad
bkt701">', '<!-- m2 template 0 -->']
close tags:
['</title>', '</script>', '</script>']
```

```
self-closing tags:
['<br/>', '<br/>', '<br/>']
```

2.4 点号

第 1 章讲到了各种字符组，与它相关的还有一个特殊的元字符：点号`.`。一般文档里都会提到，点号可以匹配“任意字符”，点号确实可以匹配“任意字符”，常见的数字、字母、各种符号都可以，如例 2-10 所示。

例 2-10 点号.的匹配

```
re.search(r"^.$", "a") != None       #  => True
re.search(r"^.$", "0") != None       #  => True
re.search(r"^.$", "*") != None       #  => True
```

有一个字符不能由点号匹配，就是换行符`\n`。这个字符平时看不见，却存在，而且在处理时并不能忽略（第 3 章会给出具体的例子）。

如果非要匹配“任意字符”，有两种办法：可以指定使用单行匹配模式，在这种模式下，点号可以匹配换行符（☞86）；或者使用第 1 章介绍的“自制”通配字符组`[\s\S]`（也可以使用`[\d\D]`或`[\w\W]`），正好涵盖了所有字符。例 2-11 清楚地说明，这两个办法都可以匹配换行符。

例 2-11 换行符的匹配

```
re.search(r"^.$", "\n") != None          #  => False
#单行模式
re.search(r"(?s)^.$", "\n") != None      #  => True
#自制“通配字符组”
re.search(r"^[\s\S]$", "\n") != None     #  => True
```

2.5 滥用点号的问题

因为点号能匹配几乎所有的字符，所以实际应用中许多人图省事，随意使用`.*`或`.+`，结果却事与愿违，下面以双引号字符串为例来说明。

之前我们使用表达式`"[^"]*"`匹配双引号字符串，而“图省事”的做法是`".*"`。通常这么用是没有问题的，但也可能有意外，例 2-12 就是一种“出乎意料”的情况。

例 2-12 “图省事”的意外结果

```
#字符串的值是"quoted string"
print re.search(r"\".*\"", "\"quoted string\"").group(0)
"quoted string"
```

```
#字符串的值是"quoted string" and another"
print re.search(r"\".*\"", "\"quoted string\" and another\"").group(0)
"quoted string" and another"
```

用".*"匹配双引号字符串，不但可以匹配正常的双引号字符串"quoted string"，还可以匹配格式错误的字符串"quoted string" and another"。这是为什么呢？

这个问题比较复杂，现在只简要介绍，以说明图省事导致错误的原因，更深入的原因涉及正则表达式的匹配原理，在第 8 章详细介绍。

在正则表达式".*"中，点号.可以匹配任何字符，*表示可以匹配的字符串长度没有限制，所以.*在匹配过程结束以前，每遇到一个字符（除去无法匹配的\n），.*都可以匹配，但是到底是匹配这个字符，还是忽略它，将其交给之后的"来匹配呢？

答案是，具体选择取决于所使用的量词。正则表达式中的量词分为几类，之前介绍的量词都可以归到一类，叫作**匹配优先量词**（greedy quantifier，也有人将其翻译为**贪婪量词** [1]）。匹配优先量词，顾名思义，就是在拿不准是否要匹配的时候，优先尝试匹配，并且记下这个状态，以备将来“反悔”。

来看表达式".*"对字符串"quoted string"的匹配过程。

- 一开始，"匹配"，然后轮到字符 q，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配 q，并且记录下这个状态【q 也可能是.*不应该匹配的】;
- 接下来是字符 u，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配 u，并且记录下这个状态【u 也可能是.*不应该匹配的】;

……

- 现在轮到字符 g，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配 g，并且记录下这个状态【g 也可能是.*不应该匹配的】;
- 最后是末尾的"，.*可以匹配它，也可以不匹配，因为使用了匹配优先量词，所以.*先匹配"，并且记录下这个状态【"也可能是.*不应该匹配的】。

这时候，字符串之后已经没有字符了，但正则表达式中还有"没有匹配，所以只能查询之前保存备用的状态，看看能不能退回几步，照顾"的匹配。查询到最近保存的状态是:【"也可能是.*不应该匹配的】。于是让.*“反悔”对"的匹配，把"交给"，测试发现正好能匹配，所以整个匹配宣告成功。这个“反悔”的过程，专业术语叫作**回溯**（backtracking），具体的过程如图 2-1 所示。

[1] 许多文档都翻译为“贪婪量词”，单独来看这是没问题的，但考虑到正则表达式中还有其他类型的量词，其英文名字的形式较为统一，所以我在翻译《精通正则表达式》时采用了“匹配优先/忽略优先/占有优先”的名字，未见读者提出反对，故此处延用此译法。

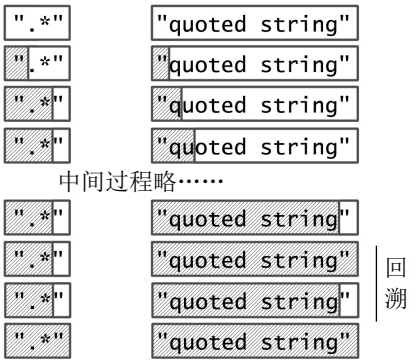


图 2–1 表达式".*"对字符串"quoted string"的匹配过程

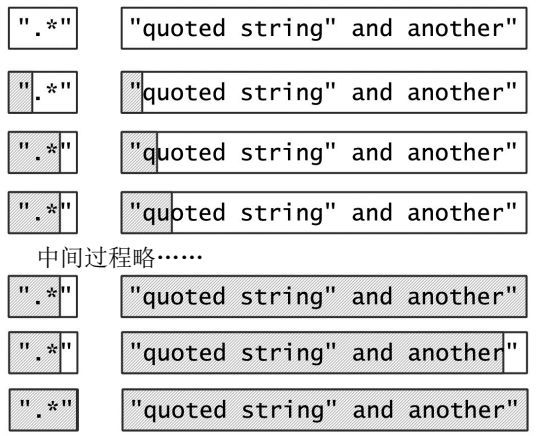


图 2–2 表达式".*"的匹配过程

如果我们把字符串换成"quoted string" and another"，.*会首先匹配第一个双引号之后的所有字符，再进行回溯，表达式中的"匹配了字符串结尾的字符"，整个匹配宣告完成，过程如图 2-2 所示。

如果要准确匹配双引号字符串，就不能图省事使用".*"，而要使用"[^"]*"，过程如图 2-3 所示。

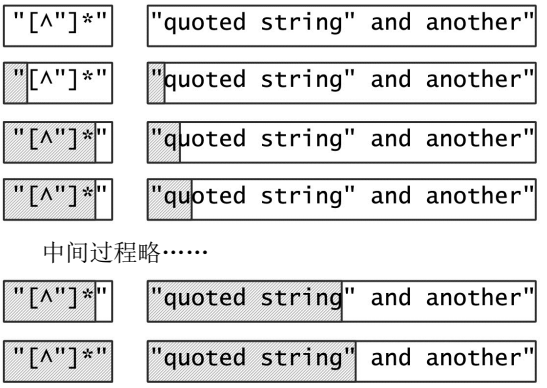


图 2–3 表达式"[^"]*"的匹配过程

2.6　忽略优先量词

也有些时候，确实需要用到`.*`（或者`[\s\S]*`），比如匹配 HTML 代码中的 JavaScript 示例就是如此。

```
<script type="text/javascript">…</script>
```

匹配的模式仍然是：匹配 open tag 和 close tag，以及它们之间的内容。open tag 是`<script type="text/javascript">`，close tag 是`</script>`，这两段的内容是固定的，非常容易写出对应的表达式，但之间的内容怎么匹配呢？在 JavaScript 代码中，各种字符都可能出现，所以不能用排除型字符组，只能用`.*` 。比如，用一个正则表达式匹配下面这段 HTML 源代码：

```
<script type="text/javascript">
alert("some punctuation <>/");
</script>
```

开头和结尾的 tag 都容易匹配，中间的代码比较麻烦，因为点号`.`不能匹配换行符，所以必须使用`[\s\S]`（或者`[\d\D]`、`[\w\W]`）。

```
<script type="text/javascript">[\s\S]*</script>
```

这个表达式确实可以匹配上面的 JavaScript 代码。但是如果遇到更复杂的情况就会出错，比如针对下面这段 HTML 代码，程序运行结果如例 2-13。

```
<script type="text/javascript">
alert("1");
</script>
<br />
<script type="text/javascript">
alert("2");
</script>
```

例 2–13　匹配 JavaScript 代码的错误

```
#假设上面的 JavaScript 代码保存在变量 htmlSource 中
jsRegex = r"<script type=\"text/javascript\">[\s\S]*</script>"
print re.search(jsRegex, htmlSource).group(0)
```

```
<script type="text/javascript">
alert("1");
</script>
<br />
```

```
<script type="text/javascript">
alert("2");
</script>
```

用`<script type="text/javascript">[\s\S]*</script>`来匹配，会一次性匹配两段 JavaScript 代码，甚至包含之间不是 JavaScript 的代码。

按照匹配原理，`[\s\S]*`先匹配所有的文本，回溯时交还最后的`</script>`，整个表达式的匹配就成功了，逻辑就是如此，无可改进。而且，这个问题也不能模仿之前双引号字符串匹配，用`[^"]*`匹配`<script…>`和`</script>`之间的代码，因为排除型字符组只能排除单个字符，`[^</script>]`不能表示“不是</script>的字符串”。

换个角度，通过改变`[\s\S]*`的匹配策略来解决问题：在不确定是否要匹配的场合，先尝试不匹配的选择，测试正则表达式中后面的元素，如果失败，再退回来尝试`.*`匹配，如此就没有问题了。

循着这个思路，正则表达式中还提供了**忽略优先量词**（lazy quantifier 或 reluctant quantifier，也有人将其翻译为**懒惰量词**），如果不确定是否要匹配，忽略优先量词会选择“不匹配”的状态，再尝试表达式中之后的元素，如果尝试失败，再回溯，选择之前保存的“匹配”的状态。

对`[\s\S]*`来说，把`*`改为`*?`就是使用了忽略优先量词，`*?`限定的元素出现次数范围与`*`完全一样，都表示“可能出现，也可能不出现，出现次数没有上限”。区别在于，在实际匹配过程中，遇到`[\s\S]`能匹配的字符，先尝试“忽略”，如果后面的元素（具体到这个表达式中，是`</script>`）不能匹配，再尝试“匹配”，这样就保证了结果的正确性，代码见例 2-14。

例 2–14 准确匹配 JavaScript 代码

```
#仍然假设 JavaScript 代码保存在变量 htmlSource 中
jsRegex = r"<script type=\"text/javascript\">[\s\S]*?</script>"
print re.search(jsRegex, htmlSource) .group(0)

<script type="text/javascript">
alert("1");
</script>

#甚至也可以逐次提取出两段 JavaScript 代码
jsRegex = r"<script type=\"text/javascript\">[\s\S]*?</script>"
for jsCode in re.findall(jsRegex, htmlSource) :
    print jsCode + "\n"

<script type="text/javascript">
alert("1");
</script>

<script type="text/javascript">
```

```
alert("2");
</script>
```

从表 2-4 可以看到，匹配优先量词与忽略优先量词逐一对应，只是在对应的匹配优先量词之后添加?，两者限定的元素能出现的次数也一样，遇到不能匹配的情况同样需要回溯；唯一的区别在于，忽略优先量词会优先选择“忽略”，而匹配优先量词会优先选择“匹配”。

表 2–4　匹配优先量词与忽略优先量词

匹配优先量词	忽略优先量词	限定次数
*	*?	可能不出现，也可能出现，出现次数没有上限
+	+?	至少出现 1 次，出现次数没有上限
?	??	至多出现 1 次，也可能不出现
{m,n}	{m,n}?	出现次数最少为 *m* 次，最多为 *n* 次
{m,}	{m,}?	出现次数最少为 *m* 次，没有上限
{,n}	{,n}?	可能不出现，也可能出现，最多出现 *n* 次

忽略优先量词还可以完成许多其他功能，典型的例子就是提取代码中的 C 语言注释。

C 语言的注释有两种：一种是在行末，以//开头；另一种可以跨多行，以/*开头，以*/结束。第一种注释很好匹配，使用//.*即可，因为点号.不能匹配换行符，所以//.*匹配的就是从//直到行末的文本，注意这里使用了量词*，因为//可能就是该行最后两个字符；第二种注释稍微复杂一点，因为/*…*/的注释和 JavaScript 一样，可能分成许多段，所以必须用到忽略优先量词；同时因为注释可能横跨多行，所以必须使用[\s\S]。因此，整个表达式就是/*[\s\S]*?*/（别忘了*的转义）。

另一个比较典型的例子是提取出HTML代码中的超链接。常见的超链接形似text。它以<a开头，以结束，*href*属性是超链接的地址。我们无法预先判断<a>和之间到底会出现哪些字符，不会出现哪些字符，只知道其中的内容一直到结束[1]，程序代码见例 2-15。

例 2–15　提取网页中所有的超链接 tag

```
#仍然获得 yahoo 网站的源代码，存放在 htmlSource 中
for hyperlink in re.findall(r"<a\s[\s\S]+?</a>", htmlSource):
    print hyperlink
#更多结果未列出
<a href="http://search.yahoo.com/">Web</a>
```

[1] 根据 HTML 规范，<a>这个 tag 可用来表示超链接，也可以用作书签，或兼作两种用途，考虑到书签的情况很少见，这里没有做特殊处理。

```
<a href="http://images.search.yahoo.com/images">Images</a>
<a href="http://video.search.yahoo.com/video">Video</a>
```

值得注意的是，这个表达式中的`<a `之后并没有使用普通空格，而是使用字符组简记法`\s`。HTML 语法并没有规定此处的空白只能使用空格字符，也没有规定必须使用一个空白字符，所以我们用`\s`保证“至少出现一个空白字符”（但是不能没有这个空白字符，否则就不能保证匹配 tag 的 name 是 a）。

之前用来匹配 JavaScript 的表达式是`<script language="text/javascript">[\s\S]*?</script>`，它能应对的情况实在太少了：在`<script`之后可能不是空格，而是空白字符；再之后可能是`type="text/javascript"`，也可能是`type="application/javascript"`，也可能用`language`取代`type`（实际上`language`是以前的写法，现在大都用`type`），甚至可能没有属性，直接是`<script>`。[1]

所以必须改造这个表达式，将条件放宽：在 `script` 之后，可能出现空白字符，也可能直接是`>`，这部分可以用一个字符组`[\s>]`来匹配，之后的内容统一用`[\s\S]+?`匹配，忽略优先量词保证了匹配进行到最近的`</script>`为止。最终得到的表达式就是`<script[\s>][\s\S]+?</script>`。

对这个表达式稍加改造，就可以写出匹配类似 tag 的表达式。在解析页面时，常见的需求是提取表格中各行、各单元（cell）的内容。表格的 tag 是`<table>`，行的 tag 是`<tr>`，单元的 tag 是`<td>`，所以，它们可以分别用下面的表达式匹配，请注意其中的`[\s>]`，它兼顾了可能存在的其他属性（比如`<table border="1">`），同时排除了可能的错误（比如`<tablet>`）。

匹配 table	`<table[\s>][\s\S]+?</table>`
匹配 tr	`<tr[\s>][\s\S]+?</tr>`
匹配 td	`<td[\s>][\s\S]+?</td>`

在实际的 HTML 代码中，`table`、`tr`、`td` 这三个元素经常是嵌套的，它们之间存在着包含关系。但是，仅仅使用正则表达式匹配，并不能得到“某个 `table` 包含哪些 `tr`”“某个 `td` 属于哪个 `tr`”这种信息。此时需要像例 2-16 那样，用程序整理出来。

例 2–16 用正则表达式解析表格

```
# 这里用到了 Python 中的三重引号字符串，以便字符串跨越多行，细节可参考第 14 章
htmlSource = """<table>
<tr><td>1-1</td></tr>
<tr><td>2-1</td><td>2-2</td></tr>
```

[1] 严格说起来，如果只出现`<script>`，无法保证这里出现的就是 JavaScript 代码，也可能是 VBScript 代码，但考虑到真实世界中的情况，基本可以认为`<script `标识的“就是”JavaScript 代码，所以这里不进行区分。

```
</table>"""

for table in re.findall(r"<table[\s>][\s\S]+?</table>", htmlSource):
 for tr in re.findall(r"<tr[\s>][\s\S]+?</tr>", table):
     for td in re.findall(r"<td[\s>][\s\S]+?</td>", tr):
         print td,
     #输出一个换行符，以便显示不同的行
     print ""
<td>1-1</td>
<td>2-1</td> <td>2-2</td>
```

注：因为 tag 是不区分大小写的，所以如果还希望匹配大写的情况，则必须使用字符组，`table` 写成 `[tT][aA][bB][lL][eE]`，`tr` 写成 `[tT][rR]`，`td` 写成 `[tT][dD]`。

这个例子说明，正则表达式只能进行纯粹的文本处理，单纯依靠它不能整理出层次结构；如果希望解析文本的同时构建层次结构信息，则必须将正则表达式配合程序代码一起使用。

回过头想想双引号字符串的匹配，之前使用的正则表达式是`"[^"]*"`，其实也可以使用忽略优先量词解决`".*?"`（如果双引号字符串中包含换行符，则使用`"[\s\S]*?"`）。两种办法相比，哪个更好呢？

一般来说，`"[^"]*"`更好。首先，`[^"]`本身能够匹配换行符，涵盖了点号`.`可能无法应付的情况，出于习惯，很多人更愿意使用点号`.`而不是`[\s\S]`；其次，匹配优先量词只需要考虑自己限定的元素能否匹配即可，而忽略优先量词必须兼顾它所限定的元素与之后的元素，效率自然大大降低，如果字符串很长，两者的速度可能有明显的差异。

而且，有些情况下确实必须用到匹配优先量词，比如文件名的解析就如此。UNIX/Linux 下的文件名类似/usr/local/bin/python 这样，它包含两个部分：路径是/usr/local/bin/；真正的文件名是 python。为了在/usr/local/bin/python 中解析出两个部分，使用匹配优先量词是非常方便的。从字符串的起始位置开始，用`.*/`匹配路径，根据之前介绍的知识，它会回溯到最后（最右）的斜线字符/，也就是文件名之前；在字符串的结尾部分，`[^/]*`能匹配的就是真正的文件名。第 1 章介绍过`^`和`$`，它们分别表示“定位到字符串的开头”和“定位到字符串的结尾”，所以应该把`^`加在匹配路径的表达式之前，得到`^.*/`，而把`$`加在匹配真正文件名的表达式之后，得到`[^/]*$`，代码见例 2-17。

例 2–17　用正则表达式拆解 Linux/UNIX 的路径

```
print re.search(r"^.*/", "/usr/local/bin/python").group(0)
/usr/local/bin
print re.search(r"[^/]*$", "/usr/local/bin/python").group(0)
python
```

Windows 下的路径分隔符是\，比如 C:\Program Files\Python 2.7.1\python.exe，所以在正则表达式中，应该把斜线字符/换成反斜线字符\。因为在正则表达式中反斜线字符\是用来转义其他字符的，为了表示反斜线字符本身，必须连写两个反斜线，所以两个表达式分别改为^.*\\和[^\\]*$，代码见例 2-18。

例 2–18　用正则表达式拆解 Windows 的路径

```
#反斜线\必须转义写成\\
print re.search(r"^.*\\", "C:\\Program Files\\Python 2.7.1\\python.exe").group(0)
C:\Program Files\Python 2.7.1\
print re.search(r"[^\\]*$", "C:\\Program Files\\Python 2.7.1\\python.exe").group(0)
python.exe
```

2.7 转义

前面讲解了匹配优先量词和忽略优先量词，现在介绍量词的转义。[1]

在正则表达式中，*、+、?等作为量词的字符具有特殊意义，但有些情况下我们需要的就是这些字符本身，此时就必须使用转义，也就是在它们之前添加反斜线\。

对常用量词所使用的字符+、*、?来说，如果希望表示这三个字符本身，直接添加反斜线，变为\+、*、\?即可。但是在一般形式的量词{m,n}字符串中，虽然具有特殊含义的字符不止一个，转义时却只需要给第一个{添加反斜线即可。也就是说，如果希望匹配字符串{m,n}，正则表达式必须写成\{m,n}。

值得一提的还有忽略优先量词的转义，虽然忽略优先量词也包含不止一个字符，但是在转义时却不像一般形式的量词那样，只转义第一个字符即可，而需要将两个量词全部转义。举例来说，如果要匹配字符串*?，正则表达式就必须写作*\?，而不是*?，因为后者的意思是“*这个字符可能出现，也可能不出现”。

表 2-5 列出了常用量词对应字符（或字符串）的转义形式。

表 2–5　各种量词字符串的转义

量词字符/字符串	转义形式
{n}	\{n}
{m,n}	\{m,n}
{m,}	\{m,}

[1] Java 等语言还支持“占有优先量词（possessive quantifier）”，但这种量词较复杂，使用也不多，所以本书中不介绍占有优先量词。

（续表）

量词字符/字符串	转义形式
{,n}	\{,n}
*	*
+	\+
?	\?
*?	*\?
+?	\+\?
??	\?\?

之前还介绍了点号.，所以别忘了点号的转义：点号.是一个元字符，它可以匹配除换行符之外的任何字符，所以如果只想匹配点号本身，必须将它转义为\.。

因为未转义的点号可以匹配任何字符，其中也包含点号，所以经常有人忽略了对点号的转义。如果真的这样做了，在确实需要严格匹配点号时就可能出错，比如匹配小数（如 3.14）、IP 地址（如 192.168.1.1）、E-mail 地址（如 someone@somehost.com）。所以，如果要匹配的文本包含点号，一定不要忘记转义正则表达式中的点号，否则就有可能出现例 2-19 所示的错误。

例 2–19　忽略转义点号可能导致错误

```
#错误判断浮点数
print re.search(r"^\d+.\d+$", "3.14") != None      #  => True
print re.search(r"^\d+.\d+$", "3a14") != None      #  => True
#准确判断浮点数
print re.search(r"^\d+\.\d+$", "3.14") != None     #  => True
print re.search(r"^\d+\.\d+$", "3a14") != None     #  => False
```

第 3 章　括号

3.1　分组

用字符组和量词可以匹配引号字符串，也可以匹配 HTML tag，如果需要用正则表达式匹配身份证号码，依靠字符组和量词能不能做到呢？

身份证号码是一个长度为 15 或 18 个字符的字符串，如果是 15 位，则全部由数字组成，首位不能为 0；如果是 18 位，前 17 位全部是数字，首位同样不能是 0，末位可能是数字，也可能是字母x。[1] 规则非常明确，可以着手编写正则表达式了。

首位是数字，不能为 0	`[1-9]`
除去首末 2 位，剩下 13 位或 16 位，都是数字	`\d{13, 16}`
末位可能是数字，也可能是 x	`[0-9x]`

整个表达式是`[1-9]\d{13,16}[0-9x]`，它的匹配如例 3-1 所示。

例 3–1　身份证号码的匹配

```
idCardRegex = r"^[1-9]\d{13,16}[0-9x]$"
re.search(idCardRegex, "110101198001017032") != None   #  => True
re.search(idCardRegex, "1101018001017016") != None     #  => True
re.search(idCardRegex, "11010119800101701x") != None   #  => True
```

看来，果然能够匹配各种形式的身份证号码，应该没问题。不过这还不够，这个正则表达式应该保证身份证号码的字符串能够匹配，其他字符串不能够匹配，例 3-2 展示了身份证号码错误匹配的情况。

例 3–2　身份证号码的错误匹配

```
re.search(idCardRegex, "1101011980010176") != None     #  => True
re.search(idCardRegex, "110101800101701x") != None     #  => True
```

[1] 一般来说，最后的 x 可以是小写也可以是大写，但也有些部门规定身份证号码最后的 x 必须是大写 X，这里为讲解方便，只考虑了小写 x 的情况；如果要兼容大写 X 或保证只能出现大写 X，只需要修改最后的字符组[0-9x]即可。

这两个字符串分明不是身份证号码（第一个是16位长，第二个虽然是15位长，但末尾是x），却都匹配了。这是为什么呢？仔细观察所用的正则表达式，会发现两点原因：第一，`\d{13,16}`表示除去首尾两位，中间的部分长度可能在13～16之间，而不是“长度要么为13，要么为16”；第二，最后的`[0-9x]`只应该对应18位身份证号码的情况，但是在这个表达式中，它也可以对应到15位身份证号码，而15位身份证号码的末位是不能为x的！

虽然字符串的长度是可变的，但是除去第一位和最后一位，中间部分的长度必须明确指定，只能是13或者16，而不能使用量词`{13,16}`；另外，末尾一位到底是`[0-9]`（也就是`\d`）还是`[0-9x]`，取决于长度——如果长度是15位，则是`\d`；如果长度是18位，则是`[0-9x]`。两种情况分别考虑，要更加清楚一些。

15位身份证号码	`[1-9]\d{14}`
18位身份证号码	`[1-9]\d{14}\d{2}[0-9x]`

看来，只要以15位号码的匹配为基础，末尾加上可能出现的`\d{2}[0-9x]`即可。最后的`\d{2}[0-9x]`必须作为一个整体，或许不出现（15位号码），或许出现（18位号码）。量词`?`可以表示“不出现，或者出现1次”，正好合适。

但是，正则表达式`\d{2}[0-9x]?`是不行的，因为量词`?`只能限定`[0-9x]`的出现，而正则表达式`\d{2}?[0-9x]?`同样不行——即使只出现一个`[0-9x]`，也可以匹配。到底怎样才能把`\d{2}[0-9x]`作为一个整体呢？

答案是：使用括号`(…)`，把正则表达式改写为`[1-9]\d{14}(\d{2}[0-9x])?`。上一章提到过，量词限定之前元素的出现，这个元素可能是一个字符，也可能是一个字符组，还可能是一个表达式——如果把一个表达式用括号包围起来，这个元素就是括号里的表达式，括号内的表达式通常被称为“子表达式”（sub-expression）。所以，`(\d{2}[0-9x])?`就表示子表达式`\d{2}[0-9x]`作为一个整体，或许不出现，或许最多出现一次。从例3-3可以看到，这个表达式确实可以准确匹配身份证号码。

例3–3　身份证号码的准确匹配

```
idCardRegex = r"^[1-9]\d{14}(\d{2}[0-9x])?$"
#应该匹配的
re.search(idCardRegex, "110101198001017016") != None   #  => True
re.search(idCardRegex, "1101018001017016") != None     #  => True
re.search(idCardRegex, "11010119800101701x") != None   #  => True
#不应该匹配的
re.search(idCardRegex, "1101011980010176") != None     #  => False
re.search(idCardRegex, "110101800101701x") != None     #  => False
```

注：为了方便讲解，我们在正则表达式的两端添加了`^`和`$`，它们分别定位到字符串的起始位置和结束

位置，这样确保了表达式不会只匹配字符串的某个子串；如果要用表达式来提取数据，应当去掉^和$。下面的例子都遵循这条规则。

括号的这种功能叫作**分组**（grouping）。如果用量词限定出现次数的元素不是字符或者字符组，而是连续的几个字符甚至子表达式，就应该用括号将它们“编为一组”。比如，希望字符串 ab 重复出现一次以上，就应该写作`(ab)+`，此时`(ab)`成为一个整体，由量词`+`来限定；如果不用括号而直接写 `ab+`，受`+`限定的就只有 `b`。例 3-4 显示了有括号与无括号的表达式的匹配异同。

例 3–4　用括号改变量词的作用元素

```
re.search(r"^ab+$", "ab") != None        #  => True
re.search(r"^ab+$", "abb") != None       #  => True
re.search(r"^ab+$", "abab") != None      #  => False

re.search(r"^(ab)+$", "ab") != None      #  => True
re.search(r"^(ab)+$", "abb") != None     #  => False
re.search(r"^(ab)+$", "abab") != None    #  => True
```

有了分组，就可以准确表示“长度只能是 *m* 或 *n*”。比如在上面匹配身份证号码的例子中，要匹配一个长度为 13 或者 16 的数字字符串。常犯的错误是使用表达式`\d{13,16}`，看起来没问题，但长度为 14 或 15 的数字字符串同样会匹配。真正准确的做法是：首先匹配长度为 13 的数字字符串，然后匹配可能出现的长度为 3 的数字字符串，正则表达式就成了`\d{13}(\d{3})?`。

分组是非常有用的功能，因为使用正则表达式时经常会遇到并没有直接相连，但确实存在联系的部分，分组可以把这些概念上相关的部分“归拢”到一起，以免割裂，下面来看几个例子。

第 2 章使用表达式`<[^/][^>]*>`匹配 HTML 中的 open tag，比如`<table>`，但是这个表达式也会匹配 self-closing tag，比如`<br />`。如果把表达式改为`<[^/][^>]*[^/]>`，确实可以避免匹配 self-closing tag，但是因为两个排除型字符组要匹配两个字符，这个表达式又会漏过`<u>`之类的 open tag，仅仅依靠字符组和量词无法配合解决问题，必须动用括号的分组功能。

`<[^/][^>]*[^/]>`错过的只有一种情况，就是 tag name 为单个字母的情况。如果 tag name 不是单个字母，则第一个字母之后，必然会出现这样一个字符串：其中不包含>，结尾的字符并不是/。最后，才是 tag 结尾的>。像图 3-1 所示那样，将这几个元素拆开，能看得更清楚点。

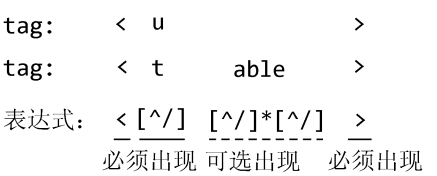


图 3–1　open tag 的准确匹配

所以，用一个括号将可选出现的部分分组，再用量词`?`限定，就可以得到兼顾这两种情况、

准确匹配 open tag 的正则表达式了，程序代码如例 3-5 所示。

例 3–5　准确匹配 open tag

```
openTagRegex = r"^<[^/]([^>]*[^/])?>$"
re.search(openTagRegex, "<u>") != None        #  => True
re.search(openTagRegex, "<table>") != None    #  => True

re.search(openTagRegex, "<u/>") != None       #  => False
re.search(openTagRegex, "</table>") != None   #  => False
```

再来看一个更复杂的例子。在 Web 服务中，经常并不希望暴露真正的程序细节，所以用某种模式的 URL 来掩盖。比如这个 URL：/foo/bar_qux.php，看起来是访问一个 PHP 页面，其实背后有复杂的路由规则。真正的结构如图 3-2 所示，foo 是模块的名称，bar 是控制器的名字，qux 则是方法名，三个名字中都只能出现小写字母。

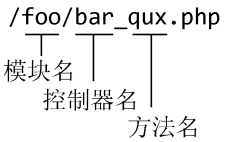


图 3–2　URL 的结构

希望能处理的情况有三种，其他情况都不予考虑。

只有模块名	/foo
只有模块名和控制器名	/foo/bar.php
模块名、控制器名和方法名同时出现	/foo/bar_qux.php

为编写通用的正则表达式来匹配，有些人是这么总结的。

最开始的斜线是必须出现的	/
模块名是必须出现的	[a-z]+
模块名之后的斜线是不一定出现的	/?
控制器名是不一定出现的	[a-z]*
控制器名之后的下画线是不一定出现的	_?
方法名也是不一定出现的	[a-z]*
最后的.php 是不一定出现的	(\.php)?

所以正则表达式就是：`/[a-z]+/?[a-z]*_?[a-z]*(\.php)?`。

仔细看看这个表达式，无论是/foo，还是/foo/bar.php，抑或是/foo/bar_qux.php，都可以匹配，看起来确实没有问题。

可是，这个表达式中只有/[a-z]+是必须出现的，其他部分都是“不一定出现”的，也就是说，其中任意一个或几个部分出现，这个表达式都可以匹配。那么，/foo/_也是可以匹配的，/foo.php 也是可以匹配的，这样就乱套了，如例 3-6 所示。

例 3–6 URL 匹配的表达式

```
urlPatternRegex = r"^/[a-z]+/?[a-z]*_?[a-z]*(\.php)?$"
re.search(urlPatternRegex, "/foo") != None                  #  => True
re.search(urlPatternRegex, "/foo/bar.php") != None          #  => True
re.search(urlPatternRegex, "/foo/bar_qux.php") != None      #  => True
re.search(urlPatternRegex, "/foo/_") != None                #  => True
re.search(urlPatternRegex, "/foo.php") != None              #  => True
```

之所以会出错，根源在于有些元素是“不一定出现”的，但它们之间却是有关联的：“不一定出现”的几个元素虽然没有前后紧密相连，却是“同生共死”的关系。这时候就要梳理清楚逻辑关系，用括号的分组功能把各种分支情况归拢到一起。

/foo 是必须出现的，之后存在两种可能：/bar.php 或者/bar_qux.php。在前一种情况中，开头的/、控制器名 bar、结尾的.php 是必须出现的；在后一种情况中，开头的/、控制器名 bar、下画线_、模块名 qux、结尾的.php 是必须出现的。

/bar.php 对应的表达式	/[a-z]+\.php
/bar_qux.php 对应的表达式	/[a-z]+_[a-z]+\.php

仔细观察这两个表达式，会发现它们可以合并：把第二个表达式中多出的部分，继续用分组括号配合量词?表示，塞到第一个表达式中，用得到的表达式配合量词?再加上最开头“必须出现”的/foo，最后得到完整的表达式。

/bar.php 对应的表达式	/[a-z]+\.php
/bar_qux.php 对应的表达式	/[a-z]+_[a-z]+\.php
合并之后的表达式	/[a-z]+(_[a-z]+)?\.php
再配合量词?	(/[a-z]+(_[a-z]+)?\.php)?
加上/foo 之后的表达式	/[a-z]+(/[a-z]+(_[a-z]+)?\.php)?

从例 3-7 可以看到，这个表达式确实杜绝了错误的匹配。

例 3–7 杜绝了错误匹配的表达式

```
urlPatternRegex = r"^/[a-z]+(/[a-z]+(_[a-z]+)?\.php)?$"
re.search(urlPatternRegex, "/foo") != None              #  => True
re.search(urlPatternRegex, "/foo/bar.php") != None      #  => True
re.search(urlPatternRegex, "/foo/bar_qux.php") != None      #  => True
```

```
re.search(urlPatternRegex, "/foo/_") != None              #  => False
re.search(urlPatternRegex, "/foo.php") != None            #  => False
```

关于括号的分组功能，最后来看 E-mail 地址的匹配：E-mail 地址以@分隔为两段，a 之前的是用户名（username），之后的是主机名（hostname），用户名一般只允许出现数字和字母（现在有些邮件服务商也允许用户名中出现点号等字符了，这种情况复杂一些，此处不做考虑），而主机名则是类似 `mail.google.com`、`mail.163.com` 之类的字符串。

用户名的匹配非常简单，其中能出现的字符主要有大写字母`[A-Z]`、小写字母`[a-z]`、阿拉伯数字字符`[0-9]`，下画线`_`、点号`.`，所以总的字符组就是`[A-Za-z0-9_.]`，又可以简化为`[\w.]`；另一方面，用户名的最大长度是 64 个字符，所以匹配用户名的正则表达式就是`[\w.]{1,64}`。

主机名匹配的情况则要麻烦一些，简单的情况比如 `somehost.com`；复杂的情况则还包括子域名，比如 `mail.somehost.net`，而且子域名可能不只一级，比如 `mail.sub.somehost.net`。查阅规范可知，主机名被点号分隔为若干段，叫作域名字段（label），每个域名字段中能出现的字符是字母字符、数字字符和横线字符，长度必须在 1~63 之间。下面看几个例子，尝试从中找到主机名的规律。

somehost.net
sub.somehost.net
mail.sub.somehost.net

看来规律是这样的：最后的域名字段是顶级域名[1]，之前的部分可以看作某种模式的重复：该模式由域名字段和点号组成，域名字段在前，点号在后。比如somehost.com就可以这么看：顶级域名是com，之前是somehost.；sub.somehost.net就可以这么看：顶级域名是net，之前是sub.和somehost.。

匹配域名字段的表达式是`[-a-zA-Z0-9]{1,63}`，匹配点号的表达式是`\.`，使用括号的分组功能，把这两个表达式分为一组，用量词`*`限定表示“不出现，或出现多次”，就得到匹配主机名的表达式`([-a-zA-Z0-9]{1,63}\.)*[-a-zA-Z0-9]{1,63}`（因为顶级域名也是一个域名字段，所以即便主机名是 `localhost`，也可以由最后那个匹配域名字段的表达式匹配）。

将匹配用户名的表达式、`@`符号、匹配主机名的表达式组合起来，就得到了完整的匹配 E-mail 地址的表达式：`[-\w.]{0,64}@([-a-zA-Z0-9]{1,63}\.)*[-a-zA-Z0-9]{1,63}`，这个表达式的匹配情况如例 3-8 所示。

[1] 虽然常见的顶级域名是 cn、com、info 之类的，最长 4 个字母，最短 2 个字母，但情况并非都是如此。在有些内部系统中，主机名中并不包含点号，可以视为顶级域名，所以这里认为顶级域名也是一个普通域名字段。

例 3-8 完整匹配 E-mail 地址的正则表达式

```
emailRegex = r"^[-\w.]{1,64}@([-a-zA-Z0-9]{1,63}\.)*[-a-zA-Z0-9]{1,63}$"
#应该匹配的
re.search(emailRegex, "abc@somehost") != None              #  => True
re.search(emailRegex, "abc@somehost.com") != None          #  => True
re.search(emailRegex, "abc@some-host.com") != None         #  => True
re.search(emailRegex, "123@somehost.info") != None         #  => True
re.search(emailRegex, "abc123@somehost.info") != None      #  => True
re.search(emailRegex, "abc123@sub.somehost.com") != None   #  => True
re.search(emailRegex, "abc123@m-s.sub.somehost.com") != None    #  => True
#不应该匹配的
re.search(emailRegex, "abc@.somehost.com") != None         #  => False
re.search(emailRegex, "a#bc@some-host.commnication") != None    #  => False
```

3.2 多选结构

之前用表达式`[1-9]\d{14}(\d{2}[0-9x])?`匹配身份证号，思路是把 18 位号码多出的 3 位“合并”到匹配 15 位号码的表达式中。这样写没有错，只是得花点心思才能理解其逻辑。

能不能直接分情况处理呢？15 位身份证号就是`[1-9]`开头，之后是 14 位数字；18 位身份证号就是`[1-9]`开头，之后是 16 位数字，最后是`[0-9x]?`。只要两个表达式中的一个能够匹配，就是合法的身份证号，这样的思路更加清晰。

15 位身份证号码	`[1-9]\d{14}`
18 位身份证号码	`[1-9]\d{14}\d{2}[0-9x]`

答案是可以的，而且仍然使用括号解决问题，只是要用到括号的另一个功能：**多选结构**（alternative）。

多选结构的形式是`(…|…)`，在括号内以竖线`|`分隔开多个子表达式，这些子表达式也叫**多选分支**（option）；在一个多选结构内，多选分支的数目没有限制。在匹配时，整个多选结构被视为单个元素，只要其中某个子表达式能够匹配，整个多选结构的匹配就能成功；如果所有子表达式都不能匹配，则整个多选结构匹配失败。

回到身份证号码匹配的例子，既然可以区分 15 位和 18 位两种情况，就可以将每种情况对应的表达式作为一个分支，“合并”为多选结构`([1-9]\d{14}|[1-9]\d{14}\d{2}[0-9x])`。这个表达式的匹配如例 3-9 所示，它同样可以准确验证身份证号码。

例 3-9 用多选结构匹配身份证号码

```
idCardRegex = r"^([1-9]\d{14}|[1-9]\d{14}\d{2}[0-9x])$"
#应该匹配的
```

```
re.search(idCardRegex, "110101198001017016") != None   #  => True
re.search(idCardRegex, "1101018001017016") != None     #  => True
re.search(idCardRegex, "11010119800101701x") != None   #  => True
#不应该匹配的
re.search(idCardRegex, "1101011980010176") != None     #  => False
re.search(idCardRegex, "110101800101701x") != None     #  => False
```

多选结构在实际中经常用到，匹配 IP 地址就是如此：IP 地址（暂不考虑 IPv6）分为 4 段（4 字节），每段都是 8 位二进制数，换算成常见的十进制，取值在 0～255 之间，中间以点号.分隔。点号.的匹配非常容易，用`\.`就可以，所以暂且忽略它，只考虑匹配这个数值的问题，而且因为 4 段 IP 地址的取值范围是相同的，只考虑其中一段的匹配即可。

要匹配十进制形式的 IP 地址，最常见的正则表达式就是`[0-9]{1,3}`，也就是 1～3 位十进制数字。粗看起来，这个表达式没什么错，细看却有很大问题。因为 256、999 这样的数值，显然不在 0~255 之间，却可以由`[0-9]{1,3}`匹配。

细致一点的表达式似乎是`[0-2][0-5][0-5]`，这样就限制了数值只能在255以内……不过，仔细想想，因为限定了第二位（十位）和第三位（个位）都只能出现 0 ～ 5 之间的字符，表达式没法匹配 168 之类的数值。

其实，问题可以这样解决：先用表达式匹配这个字符串，再将它转换为整数类型的变量 *x*，判断 *x* 是否在 0 到 255 之间：`0<=x && x<=255`。没错，这确实是一个解决问题的思路，只是有点麻烦，最好能用正则表达式“一次性”搞定这个问题。仔细想想就能发现，正则表达式虽然不能直接表示“匹配一段数值在 0～255 之间的文本”，但可以分几种情况描述这样的文本。

如果是 1 位数，那么，对数字没有限制	`[0-9]`
如果是 2 位数，那么，对数字没有限制	`[0-9]{2}`
如果是 3 位数	
如果第 1 位数字是 1，那么第 2、3 位数字都没有限制	`1[0-9][0-9]`
如果第 1 位数字是 2	
如果第 2 位数字是 0~4，那么第 3 位数字没有限制	`2[0-4][0-9]`
如果第 2 位数字是 5，那么第 3 位数字只能是 0~5	`25[0-5]`

虽然不如 `0<=x && x<=255` 判断简便，但如果文本符合其中任何一条规则（或者说，只要其中任何一个正则表达式能匹配），就可以判断它为“表示数字的数值在 0 ～ 255 之间”。用多选结构把这几条规则对应的表达式合并起来，就得到了表达式`([0-9]|[0-9]{2}|1[0-9][0-9]|2[0-4][0-9]|25[0-5])`，它的匹配如例 3-10 所示。

例 3–10　准确匹配 0~255 之间的字符串

```
partRegex = r"^([0-9]|[0-9]{2}|1[0-9][0-9]|2[0-4][0-9]|25[0-5])$"
#应该匹配的
```

```
re.search(partRegex, "0") != None    #  => True
re.search(partRegex, "98") != None   #  => True
re.search(partRegex, "168") != None  #  => True
#不应该匹配的
re.search(partRegex, "256") != None  #  => False
```

如果要更完善一点，能识别 030、005 这样的数值，可以修改对应的子表达式，为一位数和两位数的情况增加之前可能出现 0 的匹配，得到表达式`((00)?[0-9]|0?[0-9]{2}|1[0-9][0-9]|2[0-4][0-9]|25[0-5])`。

上面讲解的其实是用正则表达式匹配数值在某个范围内的字符串的通用模式，它很重要，因为许多时候会遇到类似的任务，比如匹配月（1~12）、日（不考虑只有 30 天的情况，粗略记为 1~31）、小时（0~24）、分钟（00~60）的正则表达式，用正则表达式解决这类问题，会用到同样的模式。

月	`(0?[1-9]|1[012])`
日	`(0?[1-9]|[12]\d|3[01])`
小时	`(0?[1-9]|[01]\d|2[0-4])`
分钟	`(0?[1-9]|[0-5]\d|60)`

这个模式还可以用于匹配手机号码：手机号码通常是 11 位，前面 3 位是号段，目前有 130~139 号段、150~153、155~156、180、182、185~189 号段，用多选分支`(13[0-9]|15[0-356]|18[025-9])`可以很准确地匹配号段；之后的 8 位一般没有限制，只要是数字即可，用`\d{8}`匹配。另外，手机号码开头可能有 0 或者+86，它可以用`(0|\+86)`匹配，因为整个部分是可能出现的，所以需要加上量词，也就是`(0|\+86)?`，最后得到的正则表达式就是`(0|\+86)?(13[0-9]|15[0-356]|18[025-9])\d{8}`。

多选结构还可以解决更复杂的问题，比如上一章的 tag 匹配问题，当时使用的表达式是`<[^>]+>`。一般来说，这个表达式是没有问题的，但也有可能 tag 内部还是会出现>符号，比如`<input name=txt value=">">`。这类问题使用字符组解决不了，可以使用多选结构解决。

仔细分析 tag 中可能出现>，它只可能作为属性（attribute）出现在单引号字符串和双引号字符串中，根据 HTML 规范，引号字符串中不能出现嵌套转义的引号，所以单引号字符串可以用`'[^']*'`来匹配，双引号字符串可以用`"[^"]*"`来匹配，相应的，其他内容可以用`[^'">]`来匹配，所以更完善的表达式是`<('[^']*'|"[^"]*"|[^'">])+>`。它的匹配情况见例 3-11。

例 3–11 准确的 HTML tag 匹配

```
tagRegex = r"^<('[^']*'|\"[^\"]*\"|[^'\">])+>$"
re.search(tagRegex, "<input name=txt value=\">\">") != None     #  => True
re.search(tagRegex, "<input name=txt value='>'>") != None       #  => True
re.search(tagRegex, "<a>") != None                              #  => True
```

请注意其中的量词，因为单引号字符串和双引号字符串都可以是空字符串，比如 alt=''或 alt=""，所以匹配其中文本的内容使用*；而[^'">]则没有使用量词，因为它存在于多选结构内部，多选结构外部有+量词限制，保证了它不只是匹配一个字符。如果在多选结构内部使用[^'">]*，虽然看来似乎没错，却可能导致非常奇怪的结果，不过现在不用关心，在第 143 页会给出详细讲解。

关于多选结构，最后还要补充三点。

第一，多选结构的一般表示法是(*option1*|*option2*)（其中 *option1* 和 *option2* 是两个作为多选分支的正则表达式），在多选结构中一般会同时使用括号()和竖线|；但是如果没有括号()，只出现竖线|，仍然是多选结构。从例 3-12 可以看到，ab|cd 既可以匹配 ab，也可以匹配 cd。

例 3–12　没有括号的多选结构

```
re.search(r"ab|cd", "ab") != None    #  => True
re.search(r"ab|cd", "cd") != None    #  => True
```

在多选结构中，竖线|用来分隔多选结构，而括号()用来规定整个多选结构的范围，如果没有出现括号，则将整个表达式视为一个多选结构，所以 ab|cd 等价于(ab|cd)。如果在某些地方看到没有括号的多选结构，你不用奇怪。

不过，我还是推荐明确写出两端的括号，这样更形象，也能避免一些错误。如果你仔细看，就会发现在上面的表达式中，并没有使用^和$定位字符串的起始位置和结束位置，按道理说，加上之后应该匹配更加准确，结果却并非如此。

因为竖线|的优先级很低（关于优先级，☞108），所以^ab|cd$其实是(^ab|cd$)，而不是^(ab|cd)$，它的真正意思是“字符串开头的 ab 或者字符串结尾的 cd”，而不是“只包含 ab 或 cd 的字符串”，代码见例 3-13。

例 3–13　没有括号的多选结构

```
re.search(r"^ab|cd$", "abc") != None     #  => True
re.search(r"^ab|cd$", "bcd") != None     #  => True
re.search(r"^(ab|cd)$", "abc") != None   #  => False
re.search(r"^(ab|cd)$", "bcd") != None   #  => False
```

第二，多选分支并不等于字符组。多选分支看起来类似字符组，如[abc]能匹配的字符串和(a|b|c)一样，[0-9]能匹配的字符串和(0|1|2|3|4|5|6|7|8|9)一样。从理论上说，可以完全用多选结构来替换字符组，但这种做法并不推荐，理由在于：首先，[abc]比(a|b|c)要简洁许多，在多选结构中的每个分支都必须明确写出，不能使用-范围表示法，

`(0|1|2|3|4|5|6|7|8|9)`比`[0-9]`麻烦很多；其次，在大多数情况下，`[abc]`比`(a|b|c)`的效率要高很多。所以，能用字符组解决的问题，最好不要用多选结构。

反过来，多选结构不一定能对应到字符组。因为字符组的每个“分支”的长度相同，而且只能是单个字符；而多选结构的每个“分支”的长度没有限制，甚至可以是复杂的表达式，比如`(abc|b+c*ab)`，字符组完全无能为力。

多选分支和字符组的另一点重要区别（同时也是常犯的错误）是：排除型字符组可以表示“无法由某几个字符匹配的字符”，多选结构却没有对应的结构表示“无法由某几个表达式匹配的字符串”。从例 3-14 可以看到，`[^abc]`表示“匹配除 a、b、c 之外的任意字符”，`(^a|b|c)`却不能表示“匹配除 a、b、c 之外的任意字符串”。

例 3-14 多选结构不能表示“无法由某几个表达式匹配的字符串”

```
re.search(r"(^a|b|c)", "ab") != None      #  => True
re.search(r"(^a|b|c)", "cd") != None      #  => True
```

在实际开发中确实可能遇到这种需求，不过它没有现成的解法。如果你现在就希望匹配“无法由某几个表达式匹配的字符串”，请翻到第 148 页。

第三，多选分支的排列是有讲究的。比如这个表达式`(jeff|jeffrey)`，用它匹配 **jeffrey**，结果到底是 **jeff** 还是 **jeffrey** 呢？这个问题并没有标准的答案，本书介绍的 Java、.NET、Python、Ruby、JavaScript、PHP 中，多选结构都会优先选择最左侧的分支。这一点从例 3-15 看得很清楚：如果使用的字符串是 **jeffrey**，正则表达式是`(jeff|jefferey)`还是`(Jeffrey|jeff)`，结果是不一样的（此处仅以 Python 为例，本书中介绍的其他语言中的结果与此相同）。

例 3-15 多选结构的匹配顺序

```
print re.search(r"(jeffrey|jeff)", " jeffrey").group(0)
jeffrey
print re.search(r"(jeff|jeffrey)", " jeffrey").group(0)
jeff
```

在实际开发中可能会遇到这样的情况：统计一段文本中，“湖南”和“湖南省”分别出现的次数。如果直接查找“湖南”，可能会将“湖南省”中的“湖南”也找出来，如果使用多选结构`(湖南省|湖南)`，就可以一次性找出所有“湖南”和“湖南省”，再按照字符串的长度分别计数，就可以得到两者出现的次数了。

不过，`(湖南省|湖南)`只是一个针对特殊应用的例子。在平时使用中，如果出现多选结构，应当**尽量避免多选分支中存在重复匹配**，因为这样会大大增加回溯的计算量。也就是说，应当避免这样的情况：针对多选结构`(option1|option2)`，某段文本既可以由`option1`匹配，也可以由`option2`匹配。如果出现了这样的多选结构，效率可能会受到极大影响（第 168 页总结了

可能影响效率的几种写法），尤其在受量词限定的多选结构中更是如此：一般人很难遇到`(a|[ab])`这类多选结构，但`([0-9]|\w)`之类则一不留神就会遇到。

3.3　引用分组

括号不仅能把有联系的元素归拢起来并分组，还有其他的作用——使用括号之后，正则表达式会保存每个分组真正匹配的文本，等到匹配完成后，通过 group(*num*)之类的方法“引用”分组在匹配时捕获的内容（这个方法之前已经出现过）。其中，*num* 表示对应括号的编号，括号分组的编号规则是从左向右计数，从 1 开始。因为“捕获”了文本，所以这种功能叫作**捕获分组**（capturing group）。对应的，这种括号叫作**捕获型括号**。

举一个例子，我们经常遇到诸如 2010-12-22、2011-01-03 这类表示日期的字符串，希望从中提取出年、月、日之类的信息，就可借助捕获分组来实现。在正则表达式中，每个捕获分组都有一个编号，具体情况如图 3-3 所示。

```
字符串：    2010   -   12   -   22
字符串：    2011   -   01   -   03

表达式：    (\d{4}) - (\d{2}) - (\d{2})
分组编号：     1         2         3
```

图 3–3　分组及编号

一般来说，正则表达式匹配完成之后，会得到一个表示“匹配结果”的对象，对它调用获取分组的方法，传入分组编号 *num*，就可以得到对应分组匹配的文本。第 1 章介绍过，如果匹配成功，re.search()返回一个 MatchObject 对象。如果只需要知道“是否能匹配”，判断它是否为 None 即可；但如果获取了 MatchObject 对象，可以通过对应的方法，显示匹配结果的详细信息。使用 MatchObject.group(*num*)，就可以引用正则表达式中编号为 *num* 的分组匹配的文本。从例 3-16 可以看到，通过引用编号为 1、2、3 的捕获分组，分别获得了年、月、日的信息。

例 3–16　引用捕获分组

```
print re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(1)
2010
print re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(2)
12
print re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(3)
22
```

前面说过，*num* 的编号从 1 开始。不过，也有编号为 0 的分组，它是默认存在的，对应整个表达式匹配的文本。在许多语言中，如果调用 group()方法，不给出参数 *num*，默认就等于调用

group(0)，比如 Python 就是如此，代码见例 3-17。

例 3–17 默认存在编号为 0 的分组

```
print re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(0)
2010-12-22
print re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group()
2010-12-22
```

有些正则表达式里可能包含嵌套的括号，比如在上面的例子中，除了能单独提取出年、月、日之外，再给整个表达式加上一重括号，就出现了嵌套括号，这时候括号的编号是怎样的呢？答案很简单：无论括号如何嵌套，分组的编号都是**根据开括号出现顺序来计数的**；开括号是从左向右数起第多少个开括号，整个括号分组的编号就是多少。图 3-4 举例说明了这种编号规则，具体的代码见例 3-18。

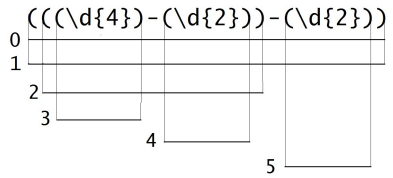


图 3–4 分组编号只取决于开括号出现的顺序

例 3–18 嵌套的括号

```
nestedGroupingRegex = r"(((\d{4})-(\d{2}))-(\d{2}))"
print re.search(nestedGroupingRegex, "2010-12-22").group(0)
2010-12-22
print re.search(nestedGroupingRegex, "2010-12-22").group(1)
2010-12-22
print re.search(nestedGroupingRegex, "2010-12-22").group(2)
2010-12
print re.search(nestedGroupingRegex, "2010-12-22").group(3)
2010
print re.search(nestedGroupingRegex, "2010-12-22").group(4)
12
print re.search(nestedGroupingRegex, "2010-12-22").group(5)
22
```

第 2 章用正则表达式`<a\s[\s\S]+?</a>`提取 HTML 中所有的超链接 tag，配合括号的分组功能，可以更进一步，依靠引用分组把超链接的地址和文本分别提取出来。通常的超链接 tag 类似这样：`<a href="url">text</a>`。其中 `url` 是超链接地址，`text` 是文本，为了准确获取这两部分内容，可以把表达式改为`<a\s+href="([^"]+)">([^<]+)</a>`。

其中给匹配 url 和 text 的表达式分别加上括号，就是`([^"]+)`和`([^<]+)`（注意其中`<a`之后是`\s+`，因为这里需要的是空白字符，而不限定是空格字符，而且可能不止一个字符）。

当然这只是最简单的情况，在等号=两端可能还有空白字符，比如`<a href = "url">text</a>`，所以正则表达式中的`=`两端也应该添加`\s*`，于是得到`<a\s+href\s*=\s*"([^"]+)">([^<]+)</a>`。

不过，属性既可以用双引号字符串表示，也可以用单引号字符串表示，比如`<a href='url'>text</a>`；甚至可以不用引号，比如`<a href=url>text</a>`。为了处理这两种情况，需要继续改造表达式：首尾出现的单引号或者双引号字符用`["']?`即可匹配；真正的 URL，既不能包含单引号，也不能包含双引号，还不能有空白字符，所以可以用`[^"'\s]+`匹配，而且这部分是需要提取出来的，别忘了它外面的括号。于是得到了最后的表达式`<a\s+href\s*=\s*["']?([^"'\s]+)["']?>([^<]+)</a>`。

现在表达式已经编写完毕，第一个括号内的表达式用来匹配 `url`，第二个括号内的表达式用来匹配 `text`，所以如果要提取 `url` 和 `text`，应该使用编号为 1 和 2 的分组。下面仍然以 yahoo.com 的首页为例来看看结果。需要说明的是，如果使用 `re.findall()`，而且正则表达式中出现了捕获型括号，那么返回数组的每个元素都是数组，其中各个元素对应各个分组的文本，所以直接用下标 2 访问得到第二个分组对应的文本，不必显式调用 `group(2)`，代码见例 3-19。

例 3–19　用分组提取出超链接的详细信息

```
# yahoo.com 的源代码已经保存在 htmlSource 中
hrefTagRegex = r"<a\s+href\s*=\s*[\"']?([^\"'\s]+)[\"']?>([^<]+)</a>"
for hyperlink in re.findall(hrefTagRegex, htmlSource):
 print hyperlink[2], hyperlink[1]
```

```
Web       http://search.yahoo.com/
Images    http://images.search.yahoo.com/images
Video     http://video.search.yahoo.com/video
……更多结果未列出
```

类似的，还可以提取出网页头部信息（`<head>`）或网页中的图片链接（`<img>`）的表达式。[1] 应当注意的是，匹配`<img>`时，在`<img`和`src`之间可能还有其他内容，比如`width=750` 之类，所以不能仅仅用`\s+`匹配，而应当添加`[^>]*?`。在`src=…`之后也同样如此。表 3-1 总结了匹配网页头部信息和图片链接的表达式。

[1] 虽然 HTML 4.0 规范中没有限制 tag 名的大小写，但 XHTML 规范要求都使用小写，在平时处理中，tag 名也大多使用小写，如果实在要处理大写的情况，可以使用字符组，比如 `img` 写作`[iI][mM][gG]`，更省事的办法是指定不区分大小写的模式，☞83。

表 3–1 提取网页头部信息和图片链接的正则表达式

任务	表达式	提取分组编号
网页头部信息	`<head>([^<]+)</head>`	1
图片链接	`<img\s+[^>]*?src=['"]?([^"'\s]+)['"]?[^>]*>`	1

应当记住的是，引用分组时，引用的是分组对应括号内的表达式捕获的文本。在这个问题上，正则表达式新手常犯错误。例 3-20 仍然是用正则表达式匹配日期字符串，两个表达式能匹配的字符串是完全相同的，引用分组的编号也是相同的，结果却不同。

例 3–20 新手容易弄错分组的结构

```
re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(1)
2010
re.search(r"(\d){4}-(\d{2})-(\d{2})", "2010-12-22").group(1)
0
```

在第一个表达式中，编号为 1 的分组对应的括号是`(\d{4})`，其中的`\d{4}`是“匹配 4 个数字字符”的子表达式。在第二个表达式中，编号为 1 的分组对应的括号是`(\d)`，其中的`\d`是“匹配 1 个数字字符”的子表达式，因为之后有量词`{4}`，所以整个括号作为单个元素，要重复出现 4 次，而且编号都是 1。于是每重复出现一次，就要更新一次匹配结果。所以在匹配过程中，编号为 1 的分组匹配的文本的值，依次是 2、0、1、0，最后的结果是 0。在实际使用时，常常有人忽略了这一细节，得到匪夷所思的匹配结果。

引用分组捕获的文本，不仅仅用于数据提取，也可以用于替换，有时候这么做非常方便。仍然举上面的日期的例子，比如希望将 YYYY-MM-DD 格式的日期变为 MM/DD/YYYY，就可以使用正则表达式替换。

在 Python 语言中进行正则表达式替换的方法是 re.sub(*pattern*, *replacement*, *string*)，其中 *pattern* 是用来匹配被替换文本的表达式，*replacement* 是要替换成的文本，*string* 是要进行替换操作的字符串，比如 re.sub(r"[a-z]", " ", *string*)就是将 *string* 中的每一个小写字母替换为一个空格。程序运行结果如例 3-21。

例 3–21 正则表达式替换

```
print re.sub(r"[a-z]", " ", "1a2b3c")
1 2 3
```

在 *replacement* 中也可以引用分组，形式是*num*，其中的 *num* 是对应分组的编号。不过，*replacement* 并不是一个正则表达式，而是一个普通字符串。根据字符串中的转义规定，\t 表制表符，\n 表示换行符，\1、\2 却不是字符串中的合法转义序列，所以也必须指定 *replacement* 为原生字符串（☞98）。例 3-22 说明了如何通过在 *replacement* 中使用引用分组转换日期字符串的格式。

例 3–22　在替换中使用分组

```
print re.sub(r"(\d{4})-(\d{2})-(\d{2})", r"\2/\3/\1", "2010-12-22")
12/22/2010
print re.sub(r"(\d{4})-(\d{2})-(\d{2})", r"\1 年\2 年\3 日", "2010-12-22")
2010 年 12 月 22 日
```

值得注意的是，如果想在 *replacement* 中引用整个表达式匹配的文本，不能使用`\0`，即便用原生字符串也不行。因为在字符串中，`\0` 开头的转义序列通常表示用八进制形式表示的字符，`\0` 本身表示 ASCII 字符编码为 0 的字符。如果一定要引用整个表达式匹配的文本，则可以稍加变通，给整个表达式加上一对括号，之后用`\1` 来引用，如例 3-23 所示。

例 3–23　在替换中，使用\1 替代\0

```
#ASCII 编码为 0 的字符无法显示
print re.sub(r"(\d{4})-(\d{2})-(\d{2})", "\\0", "2010-12-22")
print re.sub(r"(\d{4})-(\d{2})-(\d{2})", r"\0", "2010-12-22")

print re.sub(r"((\d{4})-(\d{2})-(\d{2}))", "[\\1]", "2010-12-22")
[2010-12-22]
print re.sub(r"((\d{4})-(\d{2})-(\d{2}))", r"[\1]", "2010-12-22")
2010-12-22
```

3.3.1　反向引用

英文的不少单词中都有重叠出现的字母，比如 shoot 或 beep，如果希望检查某个单词是否包含重叠出现的字母，该怎么办呢？

匹配字母的表达式是`[a-z]`（这里暂时不考虑大写的情况），所以最先想到的往往是用两个字符组`[a-z][a-z]`来匹配，但这样做并不对，因为重叠出现的字母千差万别。假设字符串是 at，a 可以由第一个`[a-z]`匹配，t 可以由第二个`[a-z]`匹配，但是因为前一个`[a-z]`和后一个`[a-z]`之间并没有联系，所以`[a-z][a-z]`其实只能匹配两个小写字母，不关心它们是否相同。

这个问题有点复杂。“重叠出现”的字母，取决于第一个`[a-z]`在运行时的匹配结果，而不能预先设定。也就是说后面的部分必须“知道”前面部分匹配的内容：如果前面的`[a-z]`匹配的是 e，后面就只能匹配 e；如果前面的`[a-z]`匹配的是 o，后面就只能匹配 o。

前面我们看到了引用分组，了解到能引用某个分组内的子表达式匹配的文本，但引用都是在匹配完成后进行的，能不能在正则表达式中引用呢？

答案是可以的，这种功能被称作**反向引用**（back-reference），它允许在正则表达式内部引用之前的捕获分组匹配的文本（也就是左侧），其形式也是`\num`，其中 *num* 表示所引用分组的编号，

编号规则与之前介绍的相同。

根据反向引用，查找连续重叠字母的表达式就是`([a-z])\1`，其中的`[a-z]`匹配第一个字母，再用括号将匹配分组，然后用`\1`来反向引用，这个表达式的匹配情况见例 3-24。

例 3–24 用反向引用匹配重复字母

```
re.search(r"^([a-z])\1$", "aa") != None       #  => True
re.search(r"^([a-z])\1$", "dd") != None       #  => True
re.search(r"^([a-z])\1$", "ac") != None       #  => False
```

在日常开发中，我们可能经常需要反向引用来建立前后联系。最常见的例子就是解析 HTML 代码时匹配 tag。之前我们说过，tag 包括 open tag 和 close tag，open tag 和 close tag 经常是成对出现的，比如`<bold>text</bold>`或`<h1>title</h1>`。

有了反向引用功能，就可以先匹配 open tag，再匹配其他内容，直到最近的 close tag 为止：在匹配 open tag 时，用一个括号分组匹配 tag name 的表达式`([^>]+)`；在匹配 close tag 时，用`\1`引用之前匹配的 tag name，就完成了配对（要注意的是，这里需要用到忽略优先量词`*?`，否则可能会出现错误匹配，理由在第 2 章匹配 JavaScript 代码时讲过）。最后得到的表达式就是`<([^>]+)>[\s\S]*?</\1>`，这个表达式的匹配如例 3-25 所示。

例 3–25 用反向引用匹配成对的 tag

```
pairedTagRegex = r"<([^>]+)>[\s\S]*?</\1>"
#应该匹配的
re.search(rpairedTagRegex, "<bold>text</bold>") != None     #  => True
re.search(rpairedTagRegex, "<h1>title</h1>") != None        #  => True
#不应该匹配的
re.search(rpairedTagRegex, "<h1>text</bold>") != None       #  => False
```

也有些 tag 更复杂一点，比如`<span class="class1">text</span>`，在 tag 名之后有一个空白字符，然后是其他属性，此时原有的表达式就无法匹配了。为应对这类情况，应当修改表达式，让分组 1 准确匹配 tag name，它可以是数字、小写字母、大写字母，所以将它修改为`<([a-zA-Z0-9]+)\s[^>]+>[\s\S]*?<\1>`，但满足了`\s[^>]+`的匹配，就无法应对之前的那些 open tag。为了兼容两种情况，必须用括号分组和量词`?`来限定，也就是改为`(\s[^>]+)?`，最后的表达式就是`<([a-zA-Z0-9]+)(\s[^>]+)?>[\s\S]*?</\1>`。具体程序如例 3-26 所示。

例 3–26 用反向引用匹配更复杂的成对 tag

```
pairedTagRegex = r"<([a-zA-Z0-9]+)(\s[^>]+)?>[\s\S]*?</\1>"
re.search(pairedTagRegex, "<bold>text</bold>") != None      #  => True
re.search(pairedTagRegex, "<h1>title</h1>") != None         #  => True
re.search(pairedTagRegex, "<span class=\"class1\">text</span>") != None  #=> True
```

```
re.search(pairedTagRegex, "<h1>text</bold>") != None        #  => False
```

反向引用还可以用在其他很多地方，比如在处理中文文本时，用它很容易找出“浩浩荡荡”“清清白白”之类 AABB，或者“如火如荼”、“越快越好”之类 AXAY 类型的四字词语。

关于反向引用，还有一点需要强调：反向引用重复的是对应捕获分组匹配的文本，而不是之前的表达式；也就是说，反向引用是一种“引用”，对应的是**由之前表达式决定的具体文本**，它**本身并不规定文本的特征**。这一点，新手常犯错误。

仍然以匹配 IP 地址为例，前面说过，IP 地址分 4 段（4 字节），匹配其中每一段的表达式是`(0{0,2}[0-9]|0?[0-9]{2}|1[0-9][0-9]|2[0-4][0-9]|25[0-5])`，之间用点号`.`分隔，所以匹配完整 IP 地址的表达式应该用量词重复这个子表达式，而不是用反向引用重复这个表达式匹配的文本。例 3-27 对比了这两个表达式，其中第二个表达式中使用了反向引用，故而要求后面 3 段与第 1 个字段完全一样，所以它只能匹配 `8.8.8.8` 之类的地址，而不能匹配 `192.168.0.1` 之类地址。

例 3–27　匹配 IP 地址的正则表达式

```
#匹配其中一段的表达式
#segment = r"(0{0,2}[0-9]|0?[0-9]{2}|1[0-9][0-9]|2[0-4][0-9]|25[0-5])"
#正确的表达式
ipAddressRegex = r"(" + segment + r"\.){3}" + segment
#错误的表达式
ipAddressRegex = segment + r"\.\1\.\1\.\1"
```

3.3.2　各种引用的记法

根据前面的介绍，对分组的引用可能出现在三种场合：在匹配完成后，用 group(*num*)之类的方法提取数据；在进行正则表达式替换时，用*num* 引用；在正则表达式内部，用*num* 引用。

不过，这只是Python语言的规定，事情并不总是如此：group(*num*)之类的方法，在各种语言中都是差不多的；但是在有些语言中，替换时引用的记法和正则表达式内部引用的记法是不同的。表 3-2 总结了各种常用语言中的两类记法。[1]

表 3–2　各种语言中引用分组的记法

语言	表达式中的反向引用	替换中的反向引用
.NET	*num*	$*num*
Java	*num*	$*num*
Objective-C	*num*	$*num*

[1] 在很多文档中都表示为*n*，但*n* 容易误解为换行符，所以本书中统一用*num* 表示。

（续表）

语言	表达式中的反向引用	替换中的反向引用
JavaScript	$num	$num
PHP	\num	\num 或$num（PHP 4.0.4 以上版本）
Python	\num	\num
Ruby	\num	\num
Golang	暂不支持	$num

看起来\num 和$num 差别不大：\1 或者$1 表示第 1 个捕获分组，\2 或者$2 表示第 2 个捕获分组……不过一般来说，$num 要好于\num。原因在于，$0 可以准确表示“第 0 个分组（也就是整个表达式匹配的文本）”，而\0 则不行，因为在不少语言的字符串中，\num 本身是一个有意义的转义序列，它表示值为 num 的 ASCII 字符，所以\0 会被解释为“ASCII 编码为 0 的字符”。但是反向引用不存在这个问题，因为不能在正则表达式还没匹配结束时，就用\0 引用整个表达式匹配的文本。

但无论是\num 还是$num，都有可能遇到二义性的问题：如果出现了\10（或者$10，这里以\num 为例），它到底表示第 10 个捕获分组\10，还是第 1 个捕获分组\1 之后跟着一个字符 0？Python 的结果见例 3-28。

例 3–28 可能具有二义性的反向引用

```
print re.sub(r"(\d)", r"\10", "123")
Traceback (most recent call last):
sre_constants.error: invalid group reference
```

原来\10 会被解释成“第 10 个捕获分组匹配的文本”，而不是“第 1 个捕获分组匹配的文本之后加上字符 0”。如果我们就是希望做到后面这步，Python 提供了\g<num>表示法，将\10 写成\g<1>0，这样同时也避免了替换时无法使用\0 的问题，代码如例 3-29 所示。

例 3–29 使用 g<n>消除二义性

```
print re.sub(r"(\d)", r"\g<1>0", "123")
102030
```

PHP 中也有专门的记法解决这类问题，在替换时可以使用\${num}的写法，准确标注所引用分组的编号，也就是说，\${1}0 表示“第 1 个捕获分组之后加上 0”，${10}表示“第 10 个捕获分组”。而$10，在第 10 个捕获分组存在的情况下，表示该捕获分组；否则，被视为空字符串。PHP 的代码见例 3-30。

例 3–30 PHP 中的引用

```
//正则表达式只包含 9 个捕获分组，将捕获的文本替换为空字符串
echo preg_replace("/^(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)/", "$10", "0123456789");
```

```
9
//正则表达式包含 10 个捕获分组，将捕获的文本替换为 10 号分组匹配的 9
echo preg_replace("/^(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)/", "$10",
"0123456789");
9
echo  preg_replace("/^(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)/",  "${1}0",
"0123456789");
00
```

注：正则表达式两端的/是分隔符，PHP 规定正则表达式两端必须使用分隔符。

Python 和 PHP 的规定比较明确，所以避免了`\num`的二义性；其他一些语言却不是如此，根据它们的文档，引用捕获分组只有`\num`（或者`$num`）一种记法，这时候`\10`（其实`\11`、`\21`等都是如此）的二义性问题就无可避免了（实际上，本书中介绍的语言，除了 Python 和 PHP 之外都是如此）。

比如 Java 对`\num`中的 *num* 是这样规定的：如果是一位数，则引用对应的捕获分组；如果是两位数且存在对应捕获分组时，引用对应的捕获分组，如果不存在对应的捕获分组，则引用一位数编号的捕获分组。

也就是说，如果确实存在编号为 10 的捕获分组，则`\10`引用此捕获分组匹配的文本；否则，`\10`表示“第 1 个捕获分组匹配的文本”和“字符 `0`”。程序的运行结果见例 3-31。

例 3–31　Java 中的引用

```
//存在 10 分组
System.out.println("0123456789".replaceAll("^(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(
\\d)(\\d)$", "$10"));
9
//不存在 10 分组
System.out.println("012345678".replaceAll("^(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\\d)(\
\d)$", "$10"));
00
```

除 Java 之外，Ruby 和 JavaScript 也采用这种规定，它看起来有点古怪，而且有一个问题无法解决：如果存在编号为 10 的捕获分组，无法用`\10`表示“编号为 1 的捕获分组和字符 0”，因为此时`\10`表示的必然是编号为 10 的捕获分组。

在开发中，尤其是进行文本替换时有时确实会遇到这个问题，它在现有的规则下是无解的。好在，一般我们并不会用到太多的捕获分组（包含捕获分组数超过 10 个的表达式很少见，也很难理解和维护）。而且，已经有越来越多的语言提供了命名分组，它可以彻底解决这个问题。

3.3.3 命名分组

捕获分组通常用数字编号来标识，但这样有几个问题：数字编号不够直观，虽然规则是“从左向右按照开括号出现的顺序计数”，但括号多了难免混淆；引用时也不够方便，上面已经讲过`\10`引起混淆的情况。

为解决这类问题，一些语言和工具提供了**命名分组**（named grouping），可以将它看作另一种捕获分组，但是标识是容易记忆和辨别的名字，而不是数字编号。

命名分组的记法也并不复杂。在 Python 中用`(?P<`*name*`>`*regex*`)`来分组的，其中的 *name* 是赋予这个分组的名字，*regex* 则是分组内的正则表达式。这样，匹配年月日的正则表达式中，可以给年、月、日的分组分别命名，再用`group(`*name*`)`来获得对应分组匹配的文本。图 3-5 说明了命名分组的结构，具体的代码见例 3-32。

```
字符串：     2010      -      12       -      22
字符串：     2011      -      01       -      03

表达式：(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})
分组名：      year             month           dat
```

图 3–5 命名分组

例 3–32 命名分组捕获

```
namedRegex = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})"
result = re.search(namedRegex, "2010-12-22")
print result.group("year")
2010
print result.group("month")
12
print result.group("day")
22
```

因为数字编号分组的历史更长，为保证向后兼容性，即便使用了命名分组，每个命名分组同时也具有数字编号，其编号规则没有变化。从例 3-33 可以看到，在全部使用命名分组的情况下，仍然可以使用数字编号来引用分组。

例 3–33 命名分组捕获时仍然保留了数字编号

```
namedRegex = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})"
result = re.search(namedRegex, "2010-12-22")
print result.group(1)
2010
print result.group(2)
12
print result.group(3)
22
```

在 Python 中，如果使用了命名分组，在表达式中反向引用时，必须使用(?P=*name*)的记法；而要进行正则表达式替换，则需要写作\g<*name*>，其中的 *name* 是分组的名字。代码见例 3-34。

例 3–34　命名分组的引用方法

```
re.search(r"^(?P<char>[a-z])(?P=char)$", "aa") != None # => True

re.sub("(?P<digit>\d)", r"\g<digit>0", "123");
102030
```

值得注意的是，命名分组不是目前通行的功能，不同语言的记法也不同，表 3-3 总结了目前常见的用法。

表 3–3　不同语言中命名分组的记法

语言	分组记法	表达式中的引用记法	替换时的引用记法
.NET	(?<*name*>…)	\k<*name*>	${*name*}
Objective-C	(?<*name*>…)	\k<*name*>	\k<*name*>
PHP	(?P<*name*>…)	(?P=*name*)[1]	不支持，只能使用\${*num*}，其中 *num* 为对应分组的数字编号
Python	(?P<*name*>…)	(?P=*name*)	\g<*name*>
JavaScript	(?<*name*>…)	\k<*name*>	$<*name*>
Ruby	(?<*name*>…)	\k<*name*>	\k<*name*>
Golang	(?P<*name*>…)	暂不支持	暂不支持

注 1：Java 5 和Java 6 都不支持命名分组，根据目前看到的JRE的文档，Java 7 开始支持命名分组[2]，其记法与.NET相同。

注 2：Ruby 1.9 以上版本才支持使用命名分组。

注 3：iOS 11 以上版本才支持使用命名分组。

注 4：支持 ES2017（TC39）的 JavaScript 才支持使用命名分组。

[1] 在 PHP 5.2.2 以后可以使用\k<*name*>或者\k'*name*'，在 PHP 5.2.4 之后可以使用\k{*name*}和\g{*name*}。

[2] 可参考 http://cr.openjdk.java.net/~sherman/6350801/webrev.00/regex/Pattern.html#groupname。

3.4 非捕获分组

到目前为止，总共介绍了括号的三种用途：**分组**，将相关的元素归拢到一起，构成单个元素；**多选结构**，规定可能出现的多个子表达式；**引用分组**，将子表达式匹配的文本存储起来，供之后引用。

这三种用途并不是彼此独立的，而是互相重叠的：单纯的分组可以视为“只包含一个多选分支的多选结构”；整个多选结构也会被视为单个元素，可以由单个量词限定。最重要的是，无论是否需要引用分组，只要出现了括号，正则表达式在匹配时就会把括号内的子表达式存储起来，提供引用。如果并不需要引用，保存这些信息无疑会影响正则表达式的性能；如果表达式比较复杂，要处理的文本又很多，更可能严重影响性能。

为解决这种问题，正则表达式提供了**非捕获分组**（non-capturing group），非捕获分组类似普通的捕获分组，只是在开括号后紧跟一个问号和冒号`(?:…)`，这样的括号叫作非捕获型括号，它只能限定量词的作用范围，不捕获任何文本。在引用分组时，分组的编号同样会按开括号出现的顺序从左到右递增，只是必须以捕获分组为准，会略过非捕获分组，如例 3-35 所示。

例 3–35 非捕获分组的使用

```
print re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(2)
12
print re.search(r"(?:\d{4})-(\d{2})-(\d{2})", "2010-12-22").group(1)
12
```

非捕获分组不需要保存匹配的文本，整个表达式的效率也因此提高，但是看起来不如捕获分组美观，所以很多人不习惯这种记法。不过，如果只需要使用括号的分组或者多选结构的功能，而没有用到引用分组，则应当尽量使用非捕获型括号。

如果不习惯这种记法，比较好的办法是，在写正则表达式时先统一使用捕获分组，确保正确之后，再把不需要引用的分组改为非捕获分组——当然，引用分组的编号可能也要调整（在上例中，只需要取月份信息，把第一个分组改为非捕获分组之后，取月份信息对应分组的编号从 2 变为 1）。

在本书中，为了使代码简洁和易于阅读，除非特殊标注，否则不管匹配完成之后是否会引用文本，都使用捕获分组。

3.5　补充

3.5.1　转义

之前讲到，如果元字符是单个出现的，直接添加反斜线字符转义即可，所以*、+、?的转义形式分别是*、\+、\?。如果元字符是成对出现的，则有可能只对第一个字符转义，比如{6}和[a-z]的转义分别是\{6}和\[a-z]。

括号的转义与它们都不同，与括号有关的所有三个元字符(、)、|都必须转义。因为括号非常重要，所以无论是开括号还是闭括号，只要出现，正则表达式就会尝试寻找整个括号，如果只转义了开括号而没有转义闭括号，一般会报告“括号不匹配”的错误。另一方面，多选结构中的|也必须转义（多选结构可以不用括号只出现|），所以，也不要忘记对|的转义，否则就可能出现例 3-36 的问题。

例 3–36　括号的转义

```
re.search(r"^\(a\)$", "(a)") != None      #  => True
re.search(r"^\(a\)$", "(a)") != None      #  => True
re.search(r"^\(a)$", "(a)") != None       #  => True
Traceback (most recent call last):
error: unbalanced parenthesis

#未转义|
re.search(r"^\(a|b\)$", "(a|b)") != None      #  => False
#同时转义了|
re.search(r"^\(a\|b\)$", "(a|b)") != None     #  => True
```

3.5.2　URL Rewrite

提到括号的分组和引用功能，就不能不提到 URL Rewrite。URL Rewrite 是常见 Web 服务器中都具备（也必需）的功能，它用来进行网址的转发，下面是一个转发的例子。

- 外部访问 URL

  ```
  http://www.example.com/blog/2006/12
  ```

- 内部实现

  ```
  http://www.example.com/blog/posts.php?year=2006&month=12
  ```

这样的好处是隔离了外部接口和内部实现，方便修改；也有利于提供更有意义、更直观的 URL。

一般来说，URL Rewrite 都是使用转发规则实现的，每条转发规则对应一类 URL，以正则表达式解析并提取出所需要的信息，重组之后再转发。比如上面的转发，就需要先提取年、月、日的信息进行重组。很自然的，我们会想到使用括号和引用分组的功能来实现。下面就以刚才提到的日期转发为例，看上面的转发规则在当前主流的 Web 服务器中如何配置。

Microsoft IIS

在 Web.config 配置文件中，找到`<rewrite>`节点，在`<rules>`下新增下面的代码。

```
<rule name="Rewrite Rule">
<match url="^blog/([0-9]{4})/([0-9]{2})/?$" />
<action type="Rewrite" url="blog/posts.php?year={R:1}&amp;month={R:2}" />
</rule>
```

其中`<match>`节点中的 url 是外部访问的 URL。对转发的 URL 而言，能接收的都是 path 部分，如果 URL 是 `http://www.example.com/blog/2006/12`，则 path 就是 `blog/2006/12`。正则表达式以`^blog`开头，分别用`[0-9]{4}`、`[0-9]{2}`匹配其中的年、月信息，因为之后的转发需要用到这些信息，所以必须使用捕获分组以便引用。另外，因为 URL 最后可能出现反斜线/，也可能不出现，意义没有区别，所以使用了量词`/?`。

Action 节点中的 url 则是转发之后（也就是内部使用）的 URL，转发到 `blog/posts.php`，且将年、月信息作为请求参数，附在后面。在 IIS 中，通过`{R:`*num*`}`的记法引用分组，其中 *num* 为对应分组的编号；另外，因为这是一个 XML 文件，所有的`&`必须转义为`&`，URL 中的`&`也不例外。

关于 IIS 中 URL Rewrite 的具体信息，可以参考下面的详细文档。

http://learn.iis.net/page.aspx/496/iis-url-rewriting-and-aspnet-routing/

Apache

在 httpd.conf 配置文件中，找到虚拟主机对应的配置字段，首先确认启用了 URL Rewrite 功能，也就是保证出现了下面这行：

```
RewriteEngine on
```

然后编写规则，上面的转义对应的规则如下：

```
RewriteRule ^blog/([0-9]{4})/([0-9]{2})/?$ blog/posts.php?year=$1&month=$2 [L]
```

以 `RewriteRule` 开头的行指定了转发规则，`RewriteRule` 之后是外部 URL 和转发的 URL，最后是可选出现的标志位（flags，`[L]`表示“如果 URL 匹配成功，按本条规则转发之后，不再考虑其他转发规则”），这几个字段之间用任意空白字符分隔。在 Apache 中，分组的引用使用`$`*num* 的形式，其中 *num* 为分组对应的编号。

关于 Apache 中 URL Rewrite 的具体信息，可以参考下面的详细文档。

Apache 2.x 版：http://httpd.apache.org/docs/2.0/misc/rewriteguide.html

Apache 1.3 版：http://httpd.apache.org/docs/1.3/mod/mod_rewrite.html

Nginx

在 Nginx.conf 配置文件中找到对应虚拟主机的配置字段，在其中添加下面的规则。

```
rewrite ^blog/([0-9]{4})/([0-9]{2})/?$ blog/posts.php?year=$1&month=$2 last;
```

以 `rewrite` 开头的行指定了转发规则，`rewrite` 之后是外部 URL 和转发的 URL，最后是可选出现的标志位（flags，`last` 的含义与 Apache 转发规则中的`[L]`相同），这几个字段之间也是用任意空白字符分隔（要注意，行的末尾必须有分号`;`）。在 Nginx 中，使用*$num* 的记法引用分组，其中 *num* 为分组对应的编号。

相对来说，Nginx 的转发功能最为强大，因为 Apache 和 IIS 的转发一般都只限于单条语句，但是 Nginx 的转发可以使用复杂的判断逻辑，比如下面的转发首先判断浏览器的 user-agent，如果是 IE 则转发，否则不转发。

```
if ($http_user_agent ~ MSIE) {
rewrite ^blog/([0-9]{4})/([0-9]{2})/?$ blog/posts.php?year=$1&month=$2 last;
}
```

关于 Nginx 中 URL Rewrite 的具体信息，可以参考下面的详细文档。

http://wiki.nginx.org/HttpRewriteModule

3.5.3　一个例子

这部分内容来自一位朋友的问题，这个问题相当有迷惑性和代表性，所以不妨列在这里，希望能解开更多读者的类似疑惑。

问题是这样的：运行 `re.findall('(\w+\.?)+', 'aaa.bbb.ccc')`，期望得到序列 `aaa.`、`bbb.`、`ccc`，实际运行的结果却只有 `ccc`，这是为什么呢？

其实答案很简单——因为表达式`(\w+\.?)+`中存在量词`+`，所以在整个正则表达式的匹配过程中，括号内的`\w+\.?`会多次匹配：第 1 次匹配 `aaa.`，第 2 次匹配 `bbb.`，第 3 次（也就是最后）匹配 `ccc`，最终这个捕获分组匹配的文本就是 `ccc`。调用 `re.findall()`时，因为存在括号（也就是捕获分组），默认返回捕获分组匹配的文本，也就是 `ccc`。

解答了这个问题之后，他继续问：如果字符串是 `aaa.bbb`，或者 `aaa.bbb.ccc.ddd`，如何能用一个表达式，逐个拆分出 `aaa.`、`bbb.`之类的子串呢？（请注意，子串的个数是变化的，并且不能预先知道。）

要搞清楚这个问题，需要记住：捕获分组的个数是**不能动态变化**的——单个正则表达式里有

多少个捕获分组，一次匹配成功之后，结果中就必然存在多少个对应的元素（捕获分组匹配的文本）。如果不能预先规定匹配结果中元素的个数，就不能使用捕获分组。如果要匹配数目不定的多段文本，必须通过重复多次匹配完成。具体到这个例子，在 `re.findall('\w+\.?', 'aaa.bbb.ccc')`中，整个正则表达式会匹配成功 3 次，得到 3 个子串；如果把字符串改为 `aaa.bbb.ccc.ddd`，则整个正则表达式会匹配成功 4 次，得到 4 个子串。

第 4 章　断言

正则表达式中的大多数结构匹配的文本会出现在最终的匹配结果中（一般用 group(0)可以得到），但是也有些结构并不真正匹配文本，而只负责判断在某个位置左/右侧的文本是否符合要求，这种结构被称为**断言**（assertion）。常见的断言有三类：单词边界、行起始/结束位置、环视。

4.1　单词边界

在文本处理中经常可能进行单词替换，比如把一段文本中的 row 都替换成 line。一般想到的是调用字符串的替换方法，直接替换 row。在不同语言中这些方法各不相同，但差别不大。

```
替换前：The row we are looking for is row 10.
替换后：The line we are looking for is line 10.
```

不过，这样替换也可能会造成意想不到的后果。

```
替换前：…tomorrow I will wear in brown standing in row 10 next to the rowdy guy…
替换后：…tomorline I will wear in blinen standing in line 10 next to the linedy guy…
```

不仅所有单词 row 都被替换成了 line，tomorrow 和 rowdy 两个单词内部的 row 也被替换成了 line，这显然不是我们想要的结果。

要解决这个问题，必须有办法确定**单词 row**，而不是**字符串 row**。为解决这类问题，正则表达式提供了专用的**单词边界**（word boundary），记为\b。它匹配的是“单词边界”位置，而不是字符。也就是说，\b 能够匹配这样的位置：一边是单词字符，另一边不是单词字符，如图 4-1 所示。

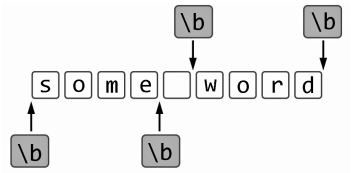


图 4–1　单词边界的匹配

表 4-1 更详细地说明了 row 配合单词边界\b 之后的匹配情况。

表 4–1 单词边界的匹配

字符串	\brow\b	\brow	row\b
tomorrow			√
brown			
row	√	√	√
rowdy		√	
表达式说明	只能是单词 row	\b 的右侧是单词字符，所以左侧不能是单词字符	\b 的左侧是单词字符，所以右侧不能是单词字符

观察表格，可以发现两点：第一，单词边界**并不区分左右**，在“单词边界”上，可能只有左侧是单词字符，也可能只有右侧是单词字符，总的来说，单词字符只能出现在一侧；第二，单词字符要求“**另一边不是单词字符**”，而不是“**另一边的字符不是单词字符**”，也就是说，一边必须出现单词字符，另一边可以出现非单词字符，也可能没有任何字符。所以，如果字符串只包含单词 word，用\bword\b 应该是可以匹配的，虽然 w 之前和 d 之后都没有任何字符。

单词边界要求一侧必须出现单词字符，到底什么是单词字符呢？

一般情况下，“单词字符”的解释是\w 能匹配的字符。在 JavaScript、PHP、Python 2、Ruby 中，\w 只能匹配[0-9a-zA-Z_]。所以在这些语言中，\b\w+\b 能准确匹配英文单词了。给定一段文本，就可以用\b\w+\b 将所有的单词提取出来，如例 4-1，制表符、换行符都不在话下。

例 4–1 单词边界的匹配

```
print re.findall(r"\b\w+\b", "a sentence\tcontains\na lot of words")
['a', 'sentence', 'contains', 'a', 'lot', 'of', 'words']
```

在 Web 开发中，经常需要对某些单词标记高亮，一般的做法是在单词的前后加上 tag，比如和，这个功能也可以由单词边界配合正则表达式替换完成，代码见例 4-2。

例 4–2 使用单词边界，准确给单词标记高亮

```
string = "tomorrow I will wear in brown standing in row 10 next to the rowdy guy"
rowWordRegex = r"\brow\b"
print re.sub(rowWordRegex, r"<span class=\"hl\">\g<0></span>", string)
tomorrow I will wear a brown sweater standing in row <span class="hl">10</span>
next to the rowdy guy
```

但是也有些单词，\b\w+\b 是无能为力的，比如 e-mail 和 M.I.T.。因为连字符-和点号.都不能由\w 匹配，所以\b\w+\b 无法匹配 e-mail，也无法匹配 M.I.T.。如果确实希望处理 e-mail 之类的“单词”，也可以把表达式改为\b[-\w]+\b。

如果使用的是.NET 或者 Python 3，在默认情况下，\w 不仅仅等价于[0-9a-zA-Z_]，还能匹配各种语言中的“单词字符”，包括中文字符，这时\b 的使用就有很大不同，具体情况（☞122）现在不展开。

与单词边界\b对应的还有非单词边界\B，两者的关系类似\s和\S、\w和\W、\d和\D：在同一种语言中 [1]，不管\b是如何规定的，\b能匹配的位置，\B就不能匹配；\B能匹配的位置，\b就不能匹配。但是在实际使用中，\B使用频率远远少于\b，所以暂不做详细介绍。

4.2　行起始/结束位置

单词边界匹配的是某个位置而不是文本，在正则表达式中，这类匹配位置的元素叫作**锚点**（anchor），它用来“定位”到某个位置。除了刚才介绍的\b，常用的锚点还有^和$。通常来说，它们分别匹配字符串的开始位置和结束位置，所以可以用来判断“整个字符串能否由表达式匹配”。这两个锚点在第 1 章粗略介绍过，现在仔细讲解。

一般情况下 [2]，^匹配**整个字符串的起始位置**。

字符串	Some sample text
^能匹配的位置	↑

依靠^，就可以用正则表达式^Some 准确验证字符串“是否以 Some 开头”，因为^会把整个表达式的匹配“定位”在字符串的开始位置。这样，即便表达式的其他部分可以在字符串中其他位置找到匹配，整个表达式也无法匹配成功。

在某些情况下，^也可以匹配字符串内部的“行起始位置”。在讲解这种情况之前，我们先来看看行是怎么划分的。

在编辑文本时，敲回车键就输入**行终止符**（Line terminal），结束当前行，新起一行。看起来，这很好理解，然而不同平台上的行终止符其实各不相同，表 4-2 列出了常见平台下的“行终止符”（这里不说“回车”而说“行终止符”是为了避免混淆，因为\r 字符就叫“回车（符）”，而\n 字符叫作“换行符”，“这里的回车是由回车符和换行符构成的”之类说法容易引起误解）。

---

[1] 更准确的说法是，“在同一种语言的同一种匹配模式下”，因为在.NET 或 Python 中，可以设定匹配模式，指定\w 采用不同的匹配规则，继而影响\b 的匹配；另外 locale（语言环境）的设定也有可能影响到\w 和\b 的匹配，只是这种情况比较少见，所以此处不介绍。

[2] 这里说的“一般情况”是指在本书提到的各种编程语言中的默认情况，只有 Ruby 是例外，下文会详细介绍。

表 4-2 不同平台下的行终止符

平台	行终止符
UNIX/Linux	\n
Windows	\r\n
Mac OS[1]	\n

也就是说，每一行的“起始位置”，就是“行终止符”之后的那个位置，如果没有专门的符号，就要考虑各种“行终止符”。下面的例子看得更清楚，为了让换行符“可见”，我们用 NL 表示。

```
first line
second line
last line
```

它其实是下面这样，其中的 NL 可能是\n，也可能是\r\n。

```
first lineNLmiddle lineNLlast lineNL
```

如果把匹配模式设定为**多行模式**（Multiline Mode，这是一种影响元字符匹配的设定，下一章详细讲解）下，^就既可以匹配整个字符串的起始位置，也可以匹配换行符之后的位置（设定多行模式最简单的办法是在正则表达式之前加上(?m)，这里虽然出现了括号，但因为是专用于指定匹配模式，所以**不会**作为捕获分组）。

表达式	(?m)^
字符串	first lineNLmiddle lineNLlast lineNL
^能匹配的位置	↑ ↑ ↑ ↑

注意，如果字符串的末尾出现了行终止符，^也会匹配这个行终止符之后的位置。这样做是有意义的，比如要在每行开头加上特殊标注（最常见的处理是把纯文本格式转换为 HTML 格式），末尾的空行自然不应该漏掉。不过一般来说，^的主要用途是与其他子表达式配合，比如像例 4-3 那样，提取每行的第一个单词。

例 4-3 提取每行的第一个单词

```
string = "first line\nsecond line\r\n\rlast line"
lineBeginWordRegex = r"(?m)^\w+"
print re.findall(lineBeginWordRegex, string)
['first', 'second', 'last']
```

有些时候，我们无论如何也不想定位到字符串内部的行起始位置，只关心整个字符串的起始

[1] 有些资料上说 Mac OS 中的换行符是\r，这并不算错，但只有 OS 9 之前的版本中，换行符才是\r。从 OS X 开始，换行符与 UNIX/Linux 一样都是\n，考虑到 OS 9 距今已经很久了，本书不考虑\r 的情况。可参考 http://en.wikipedia.org/wiki/Newline。

位置，则可以使用\A，绝大多数工具中的正则表达式都支持这个锚点，它在任何情况下（包括多行模式下）都只匹配整个字符串的起始位置，如例 4-4 所示。

例 4–4　匹配整段文本的第一个单词

```
string = "first line\nsecond line\r\nast line"
stringBeginWordRegex = r"(?m)\A\w+"
print re.findall(stringBeginWordRegex, string)
['first']
```

“行结束位置”的情况更复杂。除去“行终止符”可能由各种字符表示的情况之外，“行结束位置”可能没有任何字符，还是上面的字符串，你猜猜它有几个行终止符？

字符串：Some sample text

可能一：Some sample text

可能二：Some sample textNL

而且其中的 NL 可能是\n，也可能是\r\n。

如果要匹配字符串的最后一个单词，不但必须考虑 NL 所对应字符的多种可能，而且要兼顾 NL 是否出现，情况更加复杂。

针对这种问题，正则表达式提供了“通吃”行结束符的锚点$，它匹配的同样是位置。通常它匹配的是整个字符串的结尾位置——如果最后是行终止符，则匹配行终止符之前的位置；否则，匹配最后一个字符之后的位置。

字符串	Some sample text
$能匹配的位置	↑
字符串	Some sample textNL
$能匹配的位置	↑

这时候，无论 NL 是什么，是否存在，都可以用表达式\w+$匹配最后一个单词，代码见例 4-5。

例 4–5　匹配整段文本的最后一个单词

```
stringEndWordRegex=r"\w+$"
string = "Some sample text"
print re.findall(stringEndWordRegex, string)
['text']
string="Some sample text\n"
print re.findall(stringEndWordRegex, string)
['text']
```

如果指定了多行模式，$会匹配每个行终止符之前的位置。最后一行的情况有点特殊：如果最后一行没有行终止符，则匹配字符串的结尾位置；否则，匹配行终止符之前的位置。

表达式	(?m)$
字符串	first lineNLmiddle lineNLlast line
$能匹配的位置	↑ ↑ ↑
字符串	first lineNLmiddle lineNLlast lineNL
$能匹配的位置	↑ ↑ ↑

如果指定了多行模式，就可以用\w+$匹配每一行的最后一个单词了，代码见例 4-6。

例 4–6 匹配每行的最后一个单词

```
string = "first line\nsecond line\r\nlast line"
lineEndWordRegex = r"(?m)\w+$"
print re.findall(lineEndWordRegex, string)
['line', 'line', 'line']
```

与$类似的还有两个特殊标记\Z 和\z，它们不受多行模式的影响，在任何情况下都匹配整个字符串的结束位置。\Z 和\的主要差别在于：\Z 等价于默认模式（非多行模式）下的$，如果字符串的末尾有行终止符，则它匹配换行符之前的位置；\z 则不管行终止符，只匹配**整个字符串的结束位置**。

字符串	first lineNLmiddle lineNLlast line
\Z 能匹配的位置	↑
\z 能匹配的位置	↑
字符串	first lineNLmiddle lineNLlast lineNL
\Z 能匹配的位置	↑
\z 能匹配的位置	↑

回过头来说^和$，这两个锚点第 1 章介绍过，从数据校验的例子可以看到，re.search(*pattern*, *string*)只表示*pattern*能否在*string*中找到匹配，但是如果*pattern*只匹配*string*中的一部分，也不会返回 None；为了验证整个 *string* 能否由 *pattern* 匹配，通常的做法是在*pattern*两端加上^和$，就像例 4-7 那样。

例 4–7 借助^和$完成数据验证

```
#没有使用^和$，只匹配子串 123456
re.search(r"\d{6}", "a123456b") != None       #  => True
#使用^和$，完整验证
re.search(r"^\d{6}$", "a123456b") != None     #  => False
re.search(r"^\d{6}$", "123456") != None       #  => True
```

最常用到数据验证的场合就是对用户提交的数据进行验证，比如在网页上，要求用户在输入框（比如密码、邮箱）中填入某些信息，加以验证。一般来说，输入框中都不能输入换行符，但如果用户使用程序来提交，接收到的值就可能包含换行符。在这种情况下，在正则表达式两端添加^和$是无法准确验证的，因为$可以匹配“结尾行终止符之前的位置”，验证时就忽略了末尾的行终止符。使用\z 替换$可以堵住这个漏洞（使用\z 时，最好把^也替换成\A，这样更符合习惯，因为一般^是和$成对出现的）。下面用 Python 为例说明这一点（因为 Python 不支持\z，但是 Python 中的\Z 等价于其他语言中的\z，所以例 4-8 使用\Z）。

例 4–8　借助\A 和\Z 完成更准确的数据验证

```
#只使用^和$验证 6 位数字字符串，可能有漏洞
re.search(r"^\d{6}$", "123456") != None       #  => True
re.search(r"^\d{6}$", "123456\n") != None     #  => True
#Python 不支持\z，它的\Z 等价于其他语言中的\z
re.search(r"\A\d{6}\Z", "123456") != None     #  => True
re.search(r"\A\d{6}\Z", "123456\n") != None   #  => False
```

类似 Python 中的\Z，JavaScript 中的$的匹配也比较特殊，JavaScript 没有提供\A、\z、\Z，只有^和$，但是 JavaScript 中的$，只能匹配字符串/行的结束位置，即便字符串末尾有换行符，也是如此，所以在验证时，可以放心使用^和$。JavaScript 代码见例 4-9。

例 4–9　JavaScript 中的验证

```
/^a$/.test("a\n");      // => False
/^a$/.test("a");        // => True
```

^和$的另一个特点是，进行正则表达式替换时并不会被替换。也就是说，在起始/结束位置进行替换，只会在起始/结束位置添加一些字符，位置本身仍然存在。使用这个特性，我们可以很方便地转换文本的格式。常见的应用是将纯文本转换为 HTML，比如将纯文本的电子文档转换成 ePub 格式，就需要如此处理。这个问题最简单的思路是，使用多行模式将^替换为 ，将$替换为 ，如例 4-10 所示，注意其中使用了多行模式，这样可以找到字符串内部文本行的开始和结束位置。

例 4–10　ˆ和$的替换

```
plainText = "line1\nline2\nline3"
print plainText
lin1
line2
line3

print re.sub(r"(?m)$", "</p>", re.sub(r"(?m)^", "<p>", plainText))
<p>lin1</p>
<p>line2</p>
<p>line3</p>
```

^和$的另一个常用功能是删去多余的空白，包括行首尾的空白和空行。因为种种原因，要处理的文本可能经常包含许多空白字符，有些出现在行首，有些出现在行尾，还可能有不少空行。比如下面这段文本（为方便识别，在行的首尾分别用「和」标识）。

```
「   begin」
「  between 」
「」
「end   」
```

如果要整理格式，需要删掉不必要的空白字符，但又不能把所有空白字符都删掉（单词与单词之间的空白字符应当保留）。所以要做的其实是删除行首和行尾的空白字符，我们先删除行首的空白字符，使用的正则表达式是(?m)^\s+。

这里必须使用多行模式，否则就只能删除整个字符串首尾的空白字符。另一方面，此处使用了量词+而不是*，因为^\s*可以不匹配任何字符，这样的“删除”没有意义。将(?m)^\s+匹配的文本替换为空字符串，就执行了删除操作（一般正则表达式应用中没有单独的“删除”操作，删除操作都是通过将文本替换为空字符串实现的）。[1] 例 4-11 展示了去除字符串行首空白字符的代码。

例 4–11　去除行首的空白字符

```
withSpaces = "   begin\n between\t\n\nend  "
beginSpaceRegex = r"(?m)^\s+"
trimmedLeadingSpace = re.sub(beginSpaceRegex, "", withSpaces)
print trimmedLeadingSpace
```

为方便观察，仍然用上面的方式显示字符串。

```
「begin」
「between 」
「end   」
```

因为\s 匹配的空白字符中包含换行符\n，所以完全是空白字符的第 3 行（连同换行符）也被删掉了。现在来删行尾的空格，使用表达式\s+$，同样要记得使用多行模式，如例 4-12 所示。

例 4–12　去除行尾的空白字符

```
#代码接上面
endSpaceRegex = r"(?m)\s+$"
trimmedEndingSpace = re.sub(endSpaceRegex, "", trimmedLeadingSpace)
print trimmedEndingSpace
```

[1] 在实际开发中，可能经常要进行这种处理，比如 Web 页编辑器为了显示缩进，会在行的开头加入很多空白字符，这些空白字符都会消耗流量，所以正式发布时，可以用(?m)^\s+将它们全部删除。

仍然用上面的方式显示字符串。

```
「begin」
「between」
「end」
```

能不能用多选结构`(^\s+|\s+$)`并列两个表达式，一步完成呢？答案是**不能**。

```
withSpaces = "   begin\n between\t\n\nend  "
spaceRegex = r"(?m)(^\s+|\s+$)"
spaceTrimmed = re.sub(spaceRegex, "", withSpaces)
print spaceTrimmed
```

```
「begin」
「betweenend」
```

不但第三行被删除，第二行和第四行也合并成一行，中间的`\t\n\n` 全部被删除了，第二行末尾没有了换行符；而真正的目的其实只是想将`\t\n\n` 替换为`\n`。仔细看看正则表达式`(^\s+|\s+$)`就可以知道，在`\s+$`中，`\s`可以匹配`\t` 和`\n`，所以`\s+$`可以匹配开始的`\t\n`，同样`^\s+`可以匹配结尾的`\n`，所以`\t\n\n` 经过两步被彻底删除了。

这个例子所用的表达式很有意思 [1]，它提醒我们用多选结构合并多个表达式时，一定要小心未曾预期的后果；有时候，分几步进行反而能省去许多麻烦，这类例子在第 9 章还要讲到。

表 4-3 总结了各种语言中`^`、`$`、`\Z` 和`\z`的匹配情况。其中 Ruby 是例外，Ruby 提供了多行模式，但这个“多行模式”其实等价于常说的“单行模式”，它的作用只是让点号`.`能够匹配换行符（下一章会详细讲解各种模式），而不影响`^`和`$`的匹配；另一方面，Ruby 默认模式已经是“多行模式”，所以`$`可以匹配字符串内部的行结束位置。

表 4–3　^和$的总结

模式	行为	.NET	Java	JS	PHP	Python	Ruby	Objective–C	Golang
默认模式	^匹配字符串起始位置	√	√	√	√	√	√	√	√
	^匹配字符串内部行起始位置						√		
	$匹配字符串结束位置	√	√	√	√	√	√	√	√
	$匹配字符串末尾行终止符之前	√	√		√	√	√	√	√
	$匹配字符串内部行结束位置						√		

[1] 实际上，《精通正则表达式》（第 3 版）也详细讲到了这个例子，有兴趣的读者可以参考该书第 199 页。

（续表）

模式	行为	.NET	Java	JS	PHP	Python	Ruby	Objective-C	Golang
支持多行模式		√	√		√	√		√	√
多行模式	^匹配字符串起始位置	√	√	√	√	√	无此模式	√	√
	^匹配字符串内部行起始位置	√	√	√	√	√	无此模式	√	√
	$匹配字符串结束位置	√	√	√	√	√	无此模式	√	√
	$匹配字符串内部行结束位置	√	√	√	√	√	无此模式	√	√
	\A 等于默认模式的^	√	√		√	只能匹配字符串的起始位置	×	√	√
	\Z 等于默认模式的$	√	√		√	只能匹配字符串的结束位置	×	√	
	\z 匹配字符串的结束位置	√	√		√	无	×	√	√

注 1：JavaScript 只支持^和$，而且$不会匹配末尾换行符之前的位置。

注 2：Ruby 默认采用了多行模式，所以其中\A、\Z、\z 能匹配的位置与其他语言中的相同，但 Ruby 中默认模式下的^和$都可以匹配文本内部行的起始/结束位置，所以\A 不等于默认模式的^，\Z 也不等于默认模式的$。

4.3 环视

前面介绍过单词边界匹配的是这样的位置：一边是单词字符，另一边不是单词字符。从另一个角度来看，它能进行这样的判断：在某个位置向左/向右看，必须出现或不能出现某类字符。有时候，这种功能非常有用。

在第 2 章，用表达式`<[^/>][^>]*>`匹配 open tag，它保证了<之后不会出现/，这样就排除了`</img>`之类的 close tag，但它也可以匹配 self-closing tag，比如`<br/>`。如果将表达式改为`<[^/][^>]*[^/]>`，又会有一个问题，在<和>中的`[^/][^>]*[^/]`，能匹配的文本至少包含

两个字符，所以它无法匹配<u>。

第 3 章用正则表达式`<[^/]([^>]*[^/])?>`解决了这个问题。但是仔细想想，也可以从另一个角度描述：在开始位置匹配<，同时要求这个<之后不能是/；然后匹配中间的文本，除非在属性（也就是引号字符串）中，否则不能出现>，且长度必须大于 1（<>不是合法的 tag）；最后匹配>，同时要求这个>之前不能是/。

这样描述更加准确，逻辑也更清晰，可以用三个子表达式分别匹配这三个部分。单独来看，开头的<、中间的内容、结尾的>都不难匹配，但必须解决一个问题：匹配开头的<时，除去找到<字符，还必须向后（向右）看看，确认字符不能是/，同时又不能真正匹配这个字符，因为<和>中间的文本是有单独的子表达式匹配的；同样，结尾>的匹配也是如此。

针对这种要求，正则表达式专门提供了**环视**（look-around）用来“停在原地，四处张望”。环视类似单词边界，在它旁边的文本需要满足某种条件，而且本身不匹配任何字符。

比如正则表达式`<(?!/)`，其中的`(?!/)`是一个环视结构，`(?!…)`是这个结构的标识，`/`才是真正的表达式，整个结构的意思是“当前位置之后（右侧），不允许出现/能匹配的文本”。看起来它和`<[^/]`类似，其实大不相同：如果`<(?!/)`匹配成功，正则表达式真正匹配完成的只有<，而不包括<之后的那个字符，这样，就能准确表示“匹配<，同时这个<之后不能是/”。

再来看表达式`(?<!/)>`，其中的`(?<!/)`也是一个环视结构，`(?<!…)`是这个结构的标识，`/`才是真正的表达式，整个结构的意思是“在当前位置之前（左侧），不允许出现/能匹配的文本”（它与上面的`(?!/)`类似，只是多了一个<，更加形象地指向左侧）。这样，就能准确地表示“匹配>，同时>之前不能是/”。

至于<和>之间的文本，在第 3 章曾讲解过，可以用`('[^']*'|"[^"]*"|[^'">])+`准确匹配。

最后，把这三个部分结合起来，得到正则表达式`<(?!/)('[^']*'|"[^"]*"|[^'">])+(?<!/)>`，它可以准确匹配 open tag，而且不会错误匹配 self-closing tag，匹配情况如图 4-2 所示，代码见例 4-13。

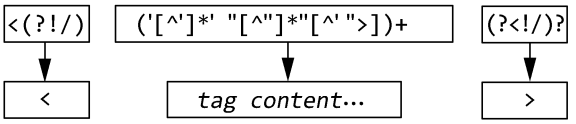


图 4–2　匹配 open tag 的表达式（注意环视结构并不会真正匹配文本）

例 4–13　使用环视结构，准确匹配 open tag

```
#注意正则表达式内部的双引号字符要转义
openTagRegex = r"\A<(?!/)('[^']*'|\"[^\"]*\"|[^'\">])+(?<!/)>\Z"
re.search(openTagRegex, "<input name=txt value=\">\">") != None     #  => True
```

```
re.search(openTagRegex, "<input name=txt value='>'>") != None       #  => True
re.search(openTagRegex, "<u>") != None                 #  => True
re.search(openTagRegex, "<br/>") != None               #  => False
re.search(openTagRegex, "<img src=\"url\"/>") != None  #  => False
```

在这个表达式中出现了两种环视：`(?!…)`和`(?<!…)`，它们的名字分别是“否定顺序环视”和“否定逆序环视”。“否定”的意思是“如果正则表达式匹配成功，则在当前位置匹配失败”，而“顺序”和“逆序”则表示正则表达式需要匹配的文本所在的位置。所以总的来说，环视一共分为 4 种：**肯定顺序环视**（positive-lookahead）、**否定顺序环视**（negative-lookahead）、**肯定逆序环视**（positive-lookbehind）、**否定逆序环视**（negative-lookbehind）。环视的分类见表 4-4。

表 4–4 环视的分类

名字	记法	判断方向	结构内表达式匹配成功的返回值
肯定顺序环视	(?=…)	向右	True
否定顺序环视	(?!…)	向右	False
肯定逆序环视	(?<=…)	向左	True
否定逆序环视	(?<!…)	向左	False

这 4 个名字容易混淆，不妨这样记忆：在当前位置，如果是朝右判断，则是顺序环视（lookahead），如果是朝左判断，则是逆序环视（lookbehind）；如果要求子表达式能匹配的字符串必须出现，则为肯定环视（positive），如果要求子表达式能匹配的字符串不能出现，则为否定环视（negative）。图 4-3 说明，对于字符串 12345，以`\d{3}`为表达式的四种环视能匹配的位置分别是：右侧必须出现三个数字字符，右侧不能出现三个数字字符，左侧必须出现三个数字字符，左侧不能出现三个数字字符。

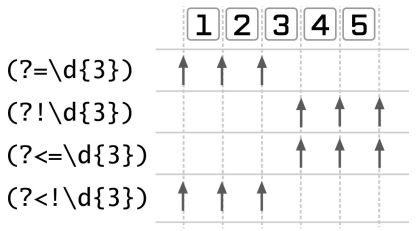


图 4–3 对字符串 12345，4 种环视能匹配的位置

环视的最大特点是“匹配完成之后还停在原地”，之前已经看到`<(?!/)`匹配的其实只有一个<字符，`(?<!/)>`匹配的也只有一个>字符，虽然它们都需要测试`/`的匹配。有时候确实需要用到“原地”的判断，因为要寻找的确实只是位置，而不需要真正匹配任何字符，比如格式化数字字符串的格式，就是如此。

英文中的数字更习惯用逗号分隔以方便阅读，比如 12345 应该写作 12,345。如果用正则表

达式来完成任务，就是“把逗号添加到这样的位置：右侧的数字字符串的长度是 3 的倍数”，看起来只需要使用肯定顺序环视就足够了，用正则表达式找到这样的位置`(?=(\d{3})+)`，将它“替换”为逗号（其实就是在这里塞进一个逗号），代码见例 4-14。

例 4–14　格式化数字字符串，第一次尝试

```
print re.sub(r"(?=(\d{3})+)", ",", "12345")
,1,2,345
```

结果却不是想象的那样。因为“右侧数字字符串”严格说应该是“当前位置右侧，所有数字字符构成的字符串”；但是`(?=(\d{3})+)`并不能表达这个意思，比如第一个字符 1 之前的位置，右侧数字字符串长度为 5，但其中存在长度为 3 的子串，所以这个位置也可以匹配（如果这样说不够清楚的话，请仔细观察图 4-3）。同样，2、3 之前的位置都是如此。

解决这个问题必须配合否定顺序环视，让`(\d{3})+`能匹配右侧的整个数字字符串，而不能只匹配其中的一个子串。也就是说，要一直匹配到“右侧不再有数字字符的位置”为止。所以，必须将表达式改写为`(?=(\d{3})+(?!\d))`，结果如例 4-15 所示。

例 4–15　格式化数字字符串，第二次尝试

```
print re.sub(r"(?=(\d{3})+(?!\d))", r",", "12345")
12,345
```

似乎没问题了，但是从例 4-16 看，如果字符串的长度正好是 3 的倍数，还是有问题。

例 4–16　格式化数字字符串，意外的情况

```
print re.sub(r"(?=(\d{3})+(?!\d))", r",", "123456")
,123,456
```

字符串的开头多出了一个逗号，因为这个位置右侧的数字字符串长度为 6。更严格地说，要加入逗号的位置其实是这样的：右侧的数字字符串的长度是 3 的倍数，且左侧也是数字字符。所以还需要加上肯定逆序环视，将正则表达式修改为`(?<=\d)(?=(\d{3})+(?!\d))`，如例 4-17 所示。

例 4–17　格式化数字字符串，最后的尝试

```
re.sub(r"(?<=\d)(?=(\d{3})+(?!\d))", r",", "123456")
123,456
```

仔细观察这个例子可以发现，表达式中其实出现了三个环视结构，其中`(?!\`是一种组合，除去这个例子中出现的嵌套和并列两种组合，环视结构还可以通过其他方式组合。关于这几类组合，本章末尾有一节专门进行讲解。

环视是非常有用的功能，日常要执行的许多操作都可能用到环视，下面再举一个例子。

我们经常会遇到中英文混排的文本，英文文本需要用空白字符来区分单词，中文文本中则很少出现空白字符。但是在转贴或格式转换时，经常会产生一些多余的空白字符：

```
中 英文混排，some English word，有多余的空 白字符
```

为了整理格式，需要删除这些空白字符。正则表达式匹配空白字符很容易，直接用`\s+`即可。但如果直接删除`\s+`能匹配的所有文本，就成了下面这样：

```
中英文混排，someEnglishword，有多余的空白字符
```

所以真正要找的，其实是这样的`\s+`：从它向左看，不能出现英文字母；从它向右看，也不能出现英文字母。所以需要在`\s+`的两端分别添加否定逆序环视和否定顺序环视，得到`(?<![a-zA-Z])\s+(?![a-zA-Z])`，结果如例 4-18 所示。

例 4–18　去掉中英文混排文本中不必要的空白字符

```
mixedString = "中 英文混排，some English word，有多余的空 白字符"
spaceBetweenChineseRegex = r"(?<![a-zA-Z])\s+(?![a-zA-Z])"
print re.sub(spaceBetweenChineseRegex, "", mixedString)
中英文混排，some English word，有多余的空白字符
```

你或许会想，这个表达式能不能改一改，比如左侧的否定逆序环视`(?<![a-zA-Z])`，能不能改为肯定环视，指定出现一个非英文字符`(?<=[^a-zA-Z])`，右侧的否定顺序环视也改为肯定顺序环视`(?=[^a-zA-Z])`？

初看起来这并没有问题，但这个问题其实涉及肯定环视和否定环视的一大根本不同：肯定环视要判断成功，字符串中**必须有字符**由环视结构中的表达式匹配；而否定环视要判断成功，却有两种情况：字符串中出现了字符，但这些字符不能由环视结构中的表达式匹配；或者字符串中**不再有任何字符**，也就是说，这个位置是字符串的起始位置或者结束位置。这两种环视的区别，可以通过例 4-19 展现。

例 4–19　使用不同的环视去掉空白字符

```
#注意字符串首尾的空白字符
mixedString = "  中 英文混排，some English word，有多余的空 白字符   "
negativeSpaceTrimRegex = r"(?<![a-zA-Z])\s+(?![a-zA-Z])"
#两端加上分隔符便于观察
print "[" + re.sub(spaceBetweenChineseRegex, "", mixedString) + "]"
[中英文混排，some English word，有多余的空白字符]
positiveSpaceTrimRegex = r"(?<=[^a-zA-Z])\s+(?=[^a-zA-Z])"
print "[" + re.sub(positiveSpaceTrimRegex, "", mixedString) + "]"
[  中英文混排，some English word，有多余的空白字符   ]
```

如果使用肯定环视，则无法去掉字符串首尾的空白。因为在字符串的开头，`\s+`虽然能匹配空白字符，但其左侧并没有任何字符，所以`(?<=[^a-zA-Z])`无法匹配成功；字符串末尾的`(?=[^a-zA-Z])`也是如此。

到现在为止，如果你觉得自己已经掌握了环视功能，来看一个更复杂的例子：在电子邮件地址中，更准确地进行主机名验证。

在上一章，我们给出了一个匹配 E-mail 地址的表达式，但它还不够完整，尤其是主机名部分（hostname）的匹配。根据规范，主机名以点号分隔为多个域名字段（label），每个域名字段可以包含大小写字母、数字字母、横线，但是横线不能出现在开头位置。关于长度，每个域名字段的长度最多为 63 个字符，整个主机名的长度最多为 255 个字符。通常用的表达式是`([-a-zA-Z0-9]{1,63}\.)*[-a-zA-Z0-9]{1,63}`，这个表达式有两个问题：第一，它允许域名字段的第一个字符是横线-；第二，它没有限定整个主机名的长度最长为 255 个字符。为准确匹配主机名，就必须解决这两个问题。

为保证域名字段的第一个字符不能是横线，可行的办法之一是单独匹配第一个字符，将表达式`[-a-zA-Z0-9]{1,63}\.`改写为`[a-zA-Z0-9][-a-zA-Z0-9]{0,62}\.`；不过，在表达式开始加上否定顺序环视更加直接，也更加自然，也就是`(?!-)[-a-zA-Z0-9]{1,63}\.`。

为保证整个主机名字符串长度小于 255 个字符，主机名中全部可能出现的字符都用`[-a-zA-Z0-9.]`表示，所以对应的肯定顺序环视就是`(?=[-a-zA-Z0-9.]{0,255})`，但是并不能直接把它添加到匹配主机名的整个表达式的开头，因为这个表达式只要求匹配一个长度在 255 个字符以内的字符串，并不能保证“之后的整个字符串长度在 255 个字符以内”。如果是单独给出一个字符串，验证它是否是合法的主机名，那么可以在这个环视中的表达式末尾添加`$`；如果是要从一长段文本中提取出某个主机名，那么主机名之后还有其他字符，只是这些字符不能是`[-a-zA-Z0-9.]`（可能是空白字符，也可能在字符串的末尾，之后没有任何字符），使用否定顺序环视`(?![-a-zA-Z0-9.])`就可以兼顾这两种情况。

最终，我们得到了更准确匹配主机名的表达式。

```
(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))((?!-)[-a-zA-Z0-9]{1,63}\.)*((?!-)[-a-zA
-Z0-9]){1,63}
```

从例 4-20 可见，这个表达式确实可以更准确地验证主机名。

例 4–20　准确匹配主机名的正则表达式

```
hostnameRegex =r "^(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))((?!-)[-a-zA-Z0-9]
{1,63}\.)*((?!-)[-a-zA-Z0-9]){1,63}$"
#应该匹配的
re.search(hostnameRegex, "localhost") != None          #  => True
re.search(hostnameRegex, "example.com") != None        #  => True
```

```
#不应该匹配的
#"e" * 64 表示将 64 个"e"连接在一起组成的字符串
re.search(hostnameRegex, "-example.com") != None       #  => False
re.search(hostnameRegex, ("e" * 64) + ".com") != None  #  => False
re.search(hostnameRegex, "e" * 256) != None            #  => False
```

4.4 补充

4.4.1 环视的价值

环视有一个很重要的用途，就是避免编写正则表达式时“牵一发动全身”的尴尬——既可以集中关注某个部分，添加复杂的限制，又不会干扰其他部分的匹配。有些时候，为添加某些限制而真正匹配文本，反而会影响整个表达式的匹配。

回想匹配 open tag 的例子：open tag 要求<之后不能出现/（否则就是 close tag，比如），而>之前不能出现/（否则就是 self-closing tag，比如）；而<和>之间的内容一般使用`[^>]+`匹配。如果使用`<[^/][^>]+[^/]>`，则`<`和`>`之间必须出现至少三个字符，即便是`<[^/][^>]*[^/]>`，`<`和`>`之间也必须至少出现两个字符，都无法匹配<u>，看来很麻烦。

如果使用环视，则非常容易解决：在`<`之后加上环视`(?!/)`，就只匹配<，同时保证匹配的<之后不是/；在`>`之前加上环视`(?<!/)`，也是如此；剩下的内容仍然由`[^>]+`匹配。这样结果清晰了很多，三个部分互不干扰，所以整个表达式就是`<(?!/)[^>]+(?<!/)>`。

环视的另一点价值在于，提取数据时杜绝错误的匹配。比如匹配邮政编码，直接的想法是找到 6 位数字构成的字符串，但仅仅用`\d{6}`提取，很可能在手机号码 13812345678、电话号码 28812506 等其他数据中找到 6 位数字构成的字符串。如果在表达式首尾添加环视，改为`(?<!\d)\d{6}(?!\d)`，就可以保证准确匹配 6 位数字构成的字符串。一般来说，凡是从文本中提取“有长度特征的数据”，都需要用到环视。

还有些时候，可以在匹配的同时以环视施加限制，达到“双管齐下”的效果。举一个例子，Java 和.NET 的正则表达式都提供了字符组运算的功能，比如匹配所有的辅音字母，最简单的思路是“从 26 个字母中减去 5 个元音字母”，在.NET 中可以写作`[a-z-[aeiou]]`，在 Java 中可以写作`[[a-z]&&[^aeiou]]`。如果使用其他语言，则没有这种便利，只能写作`[b-df-hj-np-tv-z]`，不但烦琐，而且不便理解。但是使用环视则非常简单，写作`(?![aeiou])[a-z]`，`[a-z]`真正匹配的是一个小写字母，但环视`(?![aeiou])`同时要求这个字母不能由`[aeiou]`匹配，最终效果就是“从 26 个字母中减去 5 个辅音字母”。

4.4.2　环视与分组编号

环视结构也要用到括号，这种括号是否会影响到分组编号呢？

前面说过，分组的编号只与捕获型括号有关，而不受其他任何类型括号的影响。所以，环视结构中虽然必须用到括号字符，但这里的括号只是结构需要，并不影响捕获分组，参见例 4-21。

例 4-21　单纯的环视结构并不影响引用分组

```
#表达式为(?!=ab)cd
print re.search(r"(?!=ab)(cd)", "abcd").group(0)
cd
print re.search(r"(?!=ab)(cd)", "abcd").group(1)
cd
```

前面说过，括号有多种用途，比如表示多选结构。即便括号只表示多选结构，如果没有显式指定为非捕获型括号`(?:…)`，也会被视为捕获型括号，这时候结果就大不一样了，参见例 4-22。

例 4-22　环视结构中出现了捕获型括号，会影响分组

```
#环视结构中出现了捕获型括号
print re.search(r"^(?=(ab|cd))", "abcd").group(0)

print re.search(r"^(?=(ab|cd))", "abcd").group(1)
ab
#环视结构指定使用非捕获型括号
print re.search(r"^(?=(?:ab|cd))", "abcd").group(1)
Traceback (most recent call last):
IndexError: no such group
```

这一点在实际使用中往往容易被忽略，认为环视结构并不会影响“真正”的匹配，其中的表达式匹配的文本不能从匹配结果中得到，结果算错了真正需要的分组编号。

而且，环视结构中的捕获型括号一旦匹配完成，就不能回溯。在某些情况下，这可能带来非常奇怪的问题。比如表达式`(\d+)\w+\1`，它可以匹配 123a12，其中`\d+`匹配 12，`\w+`匹配 3a，`\1` 反向引用`\d+`匹配的 12。但是，将这个表达式改为`(?<=(\d+))\w+\1`，却无法在 123a12 中找到匹配，因为一开始`\d+`就会匹配 123，之后跳出环视结构，这时`\d+`保存的备用状态全部消失了，所以无法交还 3 给`\w+`匹配，导致整个表达式匹配失败。

不过，需要用到这种表达式的情况很罕见，而且环视中能出现的表达式往往会有限制，不见得允许出现括号。

4.4.3 环视的支持程度

常用的语言（除去 Golang）大都支持环视，但语言不同，支持的程度也不同，下面详细讲解。

一般来说，所有语言都支持两种顺序环视，而且没有限制。也就是说，无论你使用肯定顺序环视，还是否定顺序环视，都可以在其中使用各种复杂的表达式。

逆序环视的情况则没有这么乐观，Ruby 1.8 中的正则表达式并不支持逆序环视，其他语言虽然支持逆序环视，但对逆序环视中的表达式能匹配的文本长度有限制[1]：Python只支持匹配固定长度文本的表达式，而Java、PHP、Objective-C只支持匹配有限长度文本的表达式，.NET和JavaScript（ES2017，或者TC39）没有任何限制。下面我们依次讲解这几种情况以及应急的解决办法。

Ruby 1.8

Ruby 1.8 不支持逆序环视。如果确实需要用到逆序环视，而且只用到肯定逆序环视，不妨采用分组来替代，比如把表达式`(?<=dog)s`改写为`dog(s)`。注意，我们给`s`添加了一个括号，如果使用`(?<=dog)s`，需要提取整个表达式匹配的文本，但使用`dog(s)`，只需要提取编号为 1 的分组捕获的文本即可。当然也可以更进一步，对`dog`使用非捕获型括号，写成`(?:dog)s`，思路与`dog(s)`相同，却不需要改动原有的分组编号。

但是，这个办法只对肯定逆序环视有效，而不能用于否定逆序环视，因为逆序环视是“从右向左”判断的，其他结构都是“从左向右”判断的，如果逆序环视中表达式能匹配的字符串长度是不固定的，就很难确定“从左向右”判断时的起点。

即便否定逆序环视中表达式能匹配的字符串长度是固定的，也难以用其他结构解决。比如表达式`(?<!dog)s`，要求 s 左侧不能出现 dog，或许你会想用表达式`(?!dog)(?:.{3})s`，但是即便字符串 s 或者 is 也无法匹配，因为这时候 s 左侧或者是空字符串，或者是只包含字母 i 的字符串，而表达式中`s`左侧的`.{3}`要求匹配三个字符。看来，表达式要改为`(?!dog)(?:.{0,3})s`，但是这时候，它又可以匹配 dogs 中的 ogs 或者 gs……总的来说，没有什么好的解决办法。

Python、Ruby 1.9

Python 规定在逆序环视中的表达式能匹配的文本长度必须是固定的。也就是说，`(?<=dog)`是合法的，`(?<=(dog|cat))`也是合法的，因为环视中的子表达式能匹配的文本都是固定的；

[1] 这些限制之所以存在，因为逆序环视的机制与正则表达式匹配的机制完全不同，它从当前位置开始，由右向左“倒过来”查找可能的匹配。实际的匹配过程更像从右向左截取一段文本，再测试它能不能由表达式匹配。如果表达式能匹配的文本长度是固定的，处理的代价就很小，否则代价可能很大。

而`(?<=dogs?)`和`(?<=(dog|cats))`则都不合法，因为环视中的子表达式能匹配的文本长度不确定。

要解决这个问题，可以用多选结构来改造表达式。比如，`dogs?`等价于`(dog|dogs)`，所以`(?<=dogs?)`可以改写为`((?<=dog)|(?<=dogs))`，同样，`(?<=(dog|cats))`可以改写为`((?<=dog)|(?<=cats))`；但是这种办法的应用场景有限，如果逆序环视中的表达式比较复杂，用多选结构列出就非常麻烦；而且一旦表达式中出现了`*`和`+`之类的量词，就根本不可能用多选结构列出了。

PHP

PHP 对逆序环视中表达式的限制要宽松点，它能匹配文本的长度可以不必限定，但是长度必须是确定的数值：`(?<=dog|cats)`是没问题的，因为两个分支的长度都是固定的；而`(?<!dogs?|cats?)`则有问题，因为两个分支的长度都是不确定的。

要解决这个问题，可以将`(?<!dogs?|cats?)`改写为`(?<!dog|dogs|cat|cats)`。但是要注意，根据 PHP 中正则表达式的规定，长度固定但不相同的多选分支，只能出现在顶级的多选结构中。也就是说，`(?<=ab(c|de))`是不支持的，而`(?<=abc|abde)`则是支持的。

Java

Java 对逆序环视中表达式的限制比 PHP 还要宽松，它能匹配的文本可以没有确定长度，但是必须有上限。所以`(?<!dogs?|cats?)`是没有问题的，甚至`(?<!ab(c|de))`也是没有问题的，但是`(?<!(dogs?){3,})`则不行，编译时会报告无法确定环视中表达式能匹配文本的最大长度。

Objective-C

Objective-C 对逆序环视中表达式的限制与 Java 相同，能匹配的文本可以没有确定长度，但是必须有上限。所以`(?<!dogs?|cats?)`是没有问题的，甚至`(?<!ab(c|de))`也是没有问题的，但是`(?<!(dogs?){3,})`则不行，编译时会报告无法确定环视中表达式能匹配文本的最大长度。

.NET

.NET 对逆序环视的支持是最为完备的。在.NET 里，逆序环视中的表达式没有任何限制，这是.NET 的正则表达式最为人称道的一点。

JavaScript

JavaScript 对环视的支持比较复杂。“经典”（从 1999 年确定的 ES3 到 2005 年确定的 ES6）的 JavaScript 只支持顺序环视，不支持逆序环视。但是，最新版本的 ES（ES2017，也叫 TC39）大幅改善了对正则表达式的支持，JavaScript 中的正则表达式也可以使用逆序环视了。考虑到

ES2017 是 2017 年定稿发布的，你读到本书的时候，流行的浏览器都已经支持 ES2017 了，所以大多数情况下应当是可以放心使用的。

尤其值得称道的是，ES2017 在制订时充分调研了业界流行的支持，没有遵循“通用”规则，约束环视结构中的子表达式能匹配的文本必须有确定长度，而是选择了与.NET 相同的标准，不对环视结构中的子表达式做任何限制。

Golang

很遗憾，到目前（Version 1.9.2）为止，Golang 内置的 Regexp 不能支持任何环视功能。如果一定要使用环视，可以考虑之前的变通办法，或者采用其他第三方的正则库。因为 Golang 的历史不长，各种特性仍然在不断增加和变化，我们期待未来 Regexp 可以直接支持环视。

总的来看，语言不同，对逆序环视的限制也不相同。逆序环视之所以麻烦，是因为其机制与正常的匹配机制完全不同：它从当前位置开始，由右向左“倒过来”查找可能的匹配。实际的操作过程更像每次从右向左截取一段文本，再判断它能不能由表达式匹配，不行再尝试……这样的过程可能要重复尝试很多次。如果表达式能匹配的文本长度确定，处理的代价就很小，否则代价可能很大。所以，比较好的做法是**尽量避免在逆序环视中使用复杂的表达式**。

4.4.4 环视的组合

环视匹配的并不是字符，而是位置。在正则表达式匹配时，环视结构匹配成功，并不会更改“当前位置”，所以多个环视可以组合在一起，实现在同一个位置的多重判断。

最常见的组合是**环视中包含环视**（逻辑如图 4-4 所示），比如之前在匹配主机名时，我们限定主机名的长度不能超过 255 个字符，使用表达式`(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))`。其中`(?![-a-zA-Z0-9.])`是包含在外层的环视中的，它要求在这个位置（也就是主机名字符串之后）不能再出现属于主机名字符串的字符，也就是保证之前的表达式匹配整个主机名字符串，而不是“可能的主机名字符串的一部分”；综合起来，`(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))`保证的是“整个主机名字符串的长度在 255 个字符以内”。

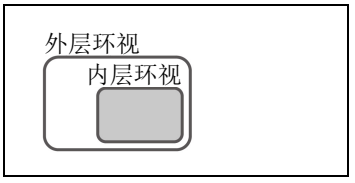


图 4–4 环视中包含环视

另一种组合是**并列多个环视**（逻辑如图 4-5 所示），它要求在当前位置，所有环视的判断都必须成功。比如要找到这样的位置：它之后是一个数字字符串，但不能是 999 开头的数字。这时候，就必须并列两个环视。表示数字字符串的表达式是`\d+`，对应的环视结构是`(?=\d+)`；表示

“不是 999 开头”的表达式的环视结构是`(?!999)`。现在要做的是把两个环视并列起来，得到`(?=\d+)(?!999)`。因为环视结构不会更改当前位置，所以先后顺序无所谓，无论是`(?=\d+)(?!999)`还是`(?!999)(?=\d+)`，效果是相同的，都要求同时满足下面两个条件：在当前位置，之后必须出现数字字符串；在当前位置，之后不能出现 999。最终的结果都是对两个环视做“与（and）”运算，也就是说，两个条件必须同时满足才算匹配成功，否则宣告当前位置匹配失败。[1]

```
#查找这样的起始位置：它之后是数字字符串，不过不能以 999 开头
#数字字符串
re.search(r"^(?=\d+)(?!999)", "1234") != None      # => True
#非数字字符串
re.search(r"^(?=\d+)(?!999)", "abcd") != None      # => False
#虽然是数字字符串，但是 999 开头
re.search(r"^(?=\d+)(?!999)", "9991234") != None   # => False
```

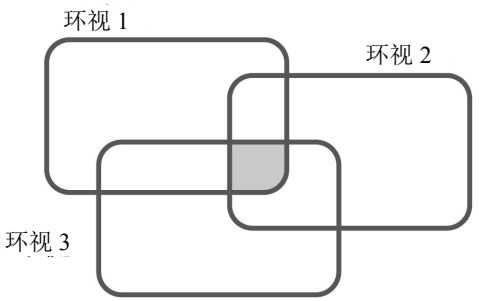


图 4–5　多个环视并列

最后一种组合是将若干个**环视作为多选分支排列在多选结构中**（逻辑如图 4-6 所示）。比如要找到这样的位置：它之后要么不是数字字符，要么是一个数字字符和一个非数字字符（比如 `1a`）。“不是数字字符”对应的环视是`(?!\d)`；而“一个数字字符和一个非数字字符”对应的环视是`(?=\d\D)`，所以总的环视就是`((?!\d)|(?=\d\D))`。虽然上一段也提过并列多个环视的组合，但在多选结构中列出多个环视结构的意义大不一样。使用多选结构时，列出的多个环视只要有一个成立，整个判断就成功；不使用多选结构时，所有列出的环视都必须成立，整个判断才成功，具体的例子可见例 4-23。

例 4–23　环视作为多选分支

```
#查找这样的起始位置，之后要么不是数字字符，要么是一个数字字符和一个非数字字符
#不是数字字符
```

[1] 如果仔细观察可能会发现，字符串 `9999` 的开始位置也不能匹配，这是因为`(?!999)`不允许出现的其实是字符串 999，而不是数值 999，如果要更准确地表示数值 999，应该使用`(?!999(?!\d))`。

```
re.search(r"^((?!\d)|(?=\d\D))", "ab") != None     # => True
#是一个数字字符和一个非数字字符
re.search(r"^((?!\d)|(?=\d\D))", "a4") != None     # => True
#单个数字字符
re.search(r"^((?!\d)|(?=\d\D))", "4") != None      # => False
#连续数字字符
re.search(r"^((?!\d)|(?=\d\D))", "44") != None     # => False
```

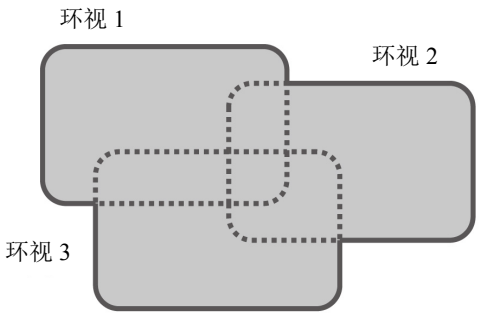


图 4–6 将环视结构排列在多选分支内

4.4.5 断言和反向引用之间的关系

断言不匹配任何字符，只匹配位置；而反向引用只引用之前的捕获分组匹配的文本，之前捕获分组中锚点表示的位置信息，在反向引用时并不会保留下来。

举例来说，如果表达式是`(\bcat\b)\s+\1`，`\1`所匹配的，就不只有单独出现的 cat，还包括单词内部的 cat（比如 cate 中的 cat），如果要验证单词 cat 是否在字符串中出现了两次，正确的做法是在反向引用两端加上单词边界`\b`，变成`(\bcat\b).*?\b\1\b`，代码见例 4-24。

例 4–24 反向引用时不会保留断言的判断

```
re.search(r"(\bcat\b).*?\1", "cat cate") != None       #  => True
#两端添加\b 之后，原有字符串无法匹配
re.search(r"(\bcat\b)\s+\b\1\b", "cat cate") != None   #   => False
re.search(r"(\bcat\b)\s+\b\1\b", "cat cat") != None    #   => True
```

有些正则表达式的资料中提到用反向引用匹配重复单词的例子，但是往往只使用`(\b\w+\b)\s+\1`来匹配“重复单词”，这其实是不对的，而应该使用`(\b\w+\b)\s+\b\1\b`。反向引用时，之前捕获分组中的断言**都会被忽略**，这一点请务必注意。

4.4.6 逆序环视的诡异之处

环视的概念确实比较难理解，所以值得多说几句。因为环视结构中的正则表达式既要尝试匹配，又不会“推进”当前的匹配位置，（通常）也不会把它匹配的文本包含在整个正则表达式的

匹配结果里。但是这还不是最麻烦的，如果你使用逆序环视，还可能遇到更诡异的现象。

出现这种现象的主要原因在于，逆序环视是“向左看”的。也就是说，它匹配的是“从当前位置向左数，最右侧的文本”。但是，通常正则表达式匹配的都是“从当前位置向右数，最左侧的文本”，所以在使用逆序环视时，环视结构中的正则表达式开始匹配的位置不同，可能决定了表达式的匹配成功与否。

如果表达式是`(?<=ab+)cd`，而字符串是 abbcd。匹配结果似乎很容易理解，分两边来看，在右侧，子表达式 `cd` 匹配字符串 cd；在左侧，字符串 abb 确实可以由子表达式 `ab+`匹配，所以整个表达式应该是能够正确匹配的。但是如果我们调整一下字符串中开始匹配的位置，把它向右边推一个字符，会怎么样？仍然分两边来看，在右侧，子表达式 `cd` 匹配字符串 cd；在左侧，字符串 bb 不能由子表达式 `ab+`匹配，所以整个表达式是不能够正确匹配的。

那么，到底哪种结果是对的？根据我的测试，无论在 JavaScript 还是在.NET 中，`(?<=ab+)cd` 都可以匹配 abbcd。看起来，正则引擎的处理办法是：遇到环视，要尽可能找到这样的位置——在它的右侧，最左侧的文本能由环视结构中的正则表达式匹配。虽然这么说有点绕，但是理解了它就能理解下面的现象：许多语言中的环视功能要求环视结构中的正则表达式匹配的文本长度必须固定，或者有明确的上限。否则，不是结果难以解释，就是实现成本高昂。

第 5 章　匹配模式

前几章已经介绍了字符组、量词、括号、断言，本章讲解正则表达式的另一个常用功能：匹配模式。

所谓**匹配模式**（match mode）[1]，指的是匹配时遵循的规则。设置特定的模式，可能会改变对正则表达式的识别，也可能会改变正则表达式中字符的匹配规定。常用的匹配模式一共有 4 种：**不区分大小写模式、单行模式、多行模式、注释模式**。

5.1　不区分大小写模式与模式的指定方式

通常，用户关心的只是文本的意义，而不是它的具体形式。比如单词 `the`，在句子中写作 `the`，在句子开头写作 `The`，还可能为了强调写作 `THE`，可是用户不管这些，只希望找到所有的 `the`。为达到这个目的，可以使用第 1 章介绍的字符组，写作`[tT][hH][eE]`，这样做确实没错，但是如果单词长一些，写起来就很麻烦了，比如 `tomorrow` 就要写成`[tT][oO][mM][oO][rR][rR][oO][wW]`。更重要的是，这样的表达式不够直观，读代码的人很难明白`[tT][oO][mM][oO][rR][rR][oO][wW]`要匹配的是 `tomorrow`。

为解决这种问题，正则表达式提供了**不区分大小写** [2]的匹配模式，指定此模式之后，在正则表达式中可以直接写`the`，就可以匹配`the`、`The`、`THE`等各种大小写形式的`the`，`tomorrow`也可以匹配各种形式的`tomorrow`，大大降低了理解的难度。

在看这个模式的应用实例之前，必须首先了解模式的指定方式。通常，有两种办法指定匹配模式：以**模式修饰符**指定，或者以**预定义的常量**作为特殊参数传入来指定。

模式修饰符即模式名称对应的单个字符，使用时将其填入特定结构`(?modifier)`中（其中

---

[1] 此处“模式”对应的英文是 mode，而不是 pattern，许多语言的文档中把正则表达式本身叫作 pattern，请注意区别。

[2] Case-Insensitive，有许多人翻译为“不敏感”，这是不对的。按照词典的解释，“敏感/对某些方面感知特别敏锐”只是 sensitive 的意思之一，sensitive 还可以表示“有能力分辨/分别”，显然 Case-Insensitive 应当取这个意思，所以翻译为“不区分大小写”。

的 *modifier* 为模式修饰符），嵌在正则表达式的开头。比如不区分大小写的匹配模式对应的模式修饰符是 `i`（case **I**nsensitive），对 `the` 指定此模式，完整的正则表达式就是 `(?i)the`。

这个表达式几乎可以原封不动地用在任何语言中（只有 JavaScript 是例外，JavaScript 不支持模式修饰符 `(?modifier)` 的记法，详见第 12 章），其意义都是相同的：对 `the` 采用不区分大小写的模式。Python 中的结果如例 5-1 所示。

例 5–1　用模式修饰符指定不区分大小写模式

```
re.search(r"(?i)the", "The") != None      #  => True
re.search(r"(?i)the", "the") != None      #  => True
re.search(r"(?i)the", "THE") != None      #  => True
```

另一种指定模式的方式是使用预定义的常量作为参数，传入正则函数。在 Python 中不区分大小写的预定义常量是 Re 类的静态成员 `re.IGNORECASE`（一般来说，它都是某个类的静态成员），在 Java 中它属于 `Pattern` 类，在.NET 中属于 `RegexOptions`，在 Ruby 中属于 `Regexp`。PHP 和 JavaScript 是例外，它们的做法（其实在 Ruby 中也可以这么做）是在正则表达式末尾的分隔符（delimeter）之后加上模式对应的字母（比如不区分大小写模式对应的字母是 `i`，则添加字母 `i`）。表 5-1 列出了常用语言中的写法。Golang 的语法更加奇怪，似乎完全不支持这种做法，只能在表达式中以修饰符来指定匹配模式。各语言中用预定义常量指定不区分大小写模式的例子参见例 5-2。

表 5–1　常用语言中不区分大小写模式的预定义常量

语言	常量
.NET	`RegexOptions.IgnoreCase`
Java	`Pattern.CASE_INSENSITIVE`
JavaScript	`/`*`regex`*`/i`
PHP	`/`*`regex`*`/i`
Python	`re.I` `re.IGNORECASE`
Ruby	`Regexp::IGNORECASE` `/`*`regex`*`/i`
Objective-C	`NSRegularExpressionCaseInsensitive`
Golang	`FoldCase`[1]

[1] Golang 虽然提供了 Flags，但没有说明如何使用。所有 Golang 的文档都建议直接使用模式修饰符指定匹配模式。

例 5–2 各语言中用预定义常量指定不区分大小写模式 [1]

.NET

```
Regex pattern = New Regex("the", RegexOptions.IgnoreCase);
pattern.IsMatch("ThE"); // => True
```

或者

```
Regex.IsMatch("the", "ThE", RegexOptions.IgnoreCase); // => True
```

Java

```
Pattern pattern = Pattern.compile("the", Pattern.CASE_INSENSITIVE);
pattern.matches("ThE").find();  // => True
```

JavaScript

```
pattern = /the/i;
pattern.test("ThE"); // => True
```

PHP =>

```
preg_match("/the/i", "ThE") != 0; // => True
```

Python

```
re.compile(r"the", re.I).search("ThE") != None #  => True
```

或者

```
re.search(r"the", "ThE", re.I) != None #  => True
```

Ruby

```
(Regexp.new("the", Regexp:: IGNORECASE) =~ "ThE") != nil #  => True
```

或者

```
(/the/i =~ "ThE") != nil #  => True
```

比较两种指定形式，模式修饰符较为通用，因为在各种语言中写法基本相同，而预定义常量在不同语言中写法不同。不过，两种形式的效果是相同的：无论以哪种方式，只要指定了不区分大小写模式，正则表达式在匹配时，就不会区分同一个字母的大小写形式（`(?i)the`和`(?i)THE`是完全等价的）。

---

[1] 在使用预定义常量指定匹配模式时，往往并不是在与匹配直接相关的函数中使用预定义常量，而是需要生成正则表达式对象，之后才能匹配。比如在 Java 的例子中，就是生成一个 Pattern 对象，然后调用该对象的对应方法（*matches*()），得到一个 Matcher 对象，再对 Matcher 对象调用 *find*() 方法，才能真正完成匹配。而之前我们看到的 Python 的例子，都是直接调用函数，将正则表达式和字符串传入（比如 re.*search*(r"the", "THE")）。Java 的这种办法，是正则表达式应用中的“面向对象式”处理；而之前看到 Python 中的方法，是正则表达式应用中的“函数式”处理，在上面的例子中，PHP 用到了此方法；而.NET、Python、Ruby、JavaScript 可以同时使用两种处理方法。关于正则表达式的处理方式，详见第 103 页。

之前看到过匹配 HTML 中 tag 的例子，比如匹配超链接 tag 的正则表达式、匹配图片 tag 及网页标题 tag 的正则表达式（☞46），虽然 XHTML 规范推荐 tag 名都用小写字母，但类似<IMG>的 tag 也很有可能出现，为同时兼容大写字母，可以使用不区分大小写模式，这样比用字符组详细列出字母的大小写形式简单很多，如表 5-2 所示。

表 5–2　采用不区分大小写模式，简化匹配 tag 的表达式

任务	表达式
提取超链接	`(?i)<a\s+href\s*=\s*["']?([^"'\s]+)["']?>([^<]+)</a>`
提取标题	`(?i)<head>([^>]+)</head>`
提取图片	`(?i)<img\s[^>]*?src=['"]?([^"']+)['"]?[^>]*>`

5.2　单行模式

元字符点号`.`**几乎**能匹配任何字符，唯有换行符`\n`是例外。但是，有时候确实需要匹配“任何字符”，比如在处理 HTML 源代码时，经常会遇到跨越多行的脚本代码。

```
<script type="text/javascript">
…code…
…code…
</script>
```

正则文档里一般都会说明“点号`.`不能匹配换行符”，不过许多人并不习惯仔细阅读文档，所以认为点号`.`能匹配任何字符，当然也就包括换行符，所以直接的想法是用`<script\s.*?</script>`来匹配。

如果你仔细读过前面的章节就会知道，因为这段 JavaScript 代码中出现了换行符，所以`.*?`的匹配最多只能延伸到第一行末尾。之前提到过，可以用`[\s\S]`之类的字符组匹配“任意字符”，所以正则表达式`<script\s[\s\S]*?</script>`能解决问题。

不过对大多数人来说，点号更自然，也更简洁，所以正则表达式提供了**单行模式**。在这种模式下，所有文本似乎只在一行里，换行符是这一行中的“普通字符”，所以可以由点号`.`匹配。

单行模式对应的模式修饰符是 s（Single line），所以如果用模式修饰符，可以在表达式的开头用`(?s)`指定，因此上面的表达式也可以改写为`(?s)<script\s.*?</script>`。使用预定义常量的写法见表 5-3。

表 5–3　常用语言中单行模式的预定义常量

语言	常量
.NET	`RegexOptions.Singleline`
Java	`Pattern.DOTALL`

（续表）

语言	常量
JavaScript	dotALL（到 ES2017 才支持）
PHP	/*regex*/s
Python	re.S re.DOTALL
Ruby	Regexp::MULTILINE /*regex*/m
Objective-C	NSRegularExpressionDotMatchesLineSeparators
Golang	DotNL（有但不能指定，原因之前解释过）

你可能注意到了，单行模式在不同语言中的称呼很不一样，甚至在同一门语言中，也可能有不同的记法（比如在 Python 中）。比如在 Java 和 Python 中叫作 DOTALL（也就是**点号通配**），这个名字确实更高明（因为常用的模式中还包含“多行模式”，它和“单行模式”没什么联系，但是这两个名字确实很迷惑人）。不过，“单行模式”成了约定俗成的称呼（通用的模式修饰符是 s，它难以联系上 DOTALL），所以本书还是沿用“单行模式”的说法。

比较奇怪的是 Ruby 的预定义常量 Multiline，它的意思是“多行”，而且它使用的模式修饰符也是 m。这确实让人费解，或许本意是“使用点号.的正则表达式可以跨越多行”？不管怎样，请一定记住，Ruby 中的“多行模式”实际上是常说的“单行模式”。

如果不想使用“点号+单行模式”的组合（比如 JavaScript 完全不支持这种模式，想用也没办法），也可以使用[\s\S]之类的字符组，它的确可以匹配任何字符，只是许多人并不习惯这样的写法，“点号+单行模式”的组合看起来更顺眼一些。

5.3 多行模式

“多行模式”听起来是与“单行模式”对应的，其实这两个模式没有任何联系。

单行模式影响的是点号的匹配规则：在默认模式下，点号.可以匹配**除换行符之外**的任何字符，在单行模式下，点号.可以匹配包括换行符在内的任何字符；**多行模式影响的是^和$的匹配规则：**在默认模式下，^和$匹配的是**整个字符串**的起始位置和结束位置，但在多行模式下，它们也能匹配**字符串内部**某一行文本的起始位置和结束位置。

假设，需要找到下面文本中所有数字字符开头的行。

```
1 line
No digit
2 line
```

解决这个问题，需要定位到每行的起始位置，尝试匹配一个数字字符，如果成功，则匹配之

后的整行文本。多行模式的模式修饰符是 m（**M**ultiline），所以在表达式的开头用(?m)指定多行模式，这样^可以定位到字符串内部每一行的起始位置；匹配数字字符的表达式是\d，因为没有指定单行模式，点号.不能匹配换行符，.*可以匹配“之后的整行文本”，整个表达式就是(?m)^\d.*，示例见例 5-3。

例 5–3　用模式修饰符指定多行模式

```
multilineString = "1 line\nNot digit\n2 line"
lineBeginWithDigitRegex = r"(?m)^\d.*"
for line in re.findall(lineBeginWithDigitRegex, multilineString):
print line
```

```
1 line
2 line
```

还可以利用多行模式下的$，给每一行的末尾添加句号，如例 5-4 所示。

例 5–4　在多行模式下，给行末添加句号

```
multilineString = "1 line\nNot digit\n2 line"
lineEndRegex = r"(?m)$"
print re.sub(lineEndRegex, ".", multilineString)
```

```
1 line.
Not digit.
2 line.
```

上面的表达式几乎是任何语言中都“通用”的，唯有 JavaScript 是例外，因为它不支持用模式修饰符指定模式，但是可以使用预定义常量来指定多行模式。在表 5-4 中列出了常用语言中预定义常量的写法。

表 5–4　常用语言中多行模式的预定义常量

语言	常量
.NET	`RegexOptions.Multiline`
Java	`Pattern.MULTILINE`
JavaScript	`/`*regex*`/m`
PHP	`/`*regex*`/m`
Python	`re.M` `re.MULTILINE`
Ruby	默认即为多行模式
Objective-C	`NSRegularExpressionAnchorsMatchLines`
Golang	没有对应的 Flags

在各种语言中，多行模式对应预定义常量的写法比较统一，都是`multiline`。值得一提的是Ruby，它默认就采用多行模式，`^`和`$`在任何情况下都能匹配文本内部的行起始/结束位置。如果要在Ruby中“摆脱”多行模式，只能用`\A`替代`^`，用`\Z`替代`$`。

5.4 注释模式

有时，用到的正则表达式可能非常复杂，不但难以编写和阅读，也难以维护。如果正则表达式也像程序源代码一样，可以添加注释，阅读和维护起来就容易多了。

为解决这个问题，许多语言支持使用`(?#comment)`的记法添加注释，*comment* 就是注释的内容。所以，上面的表达式`^\d.*?$`就可以写成这样：

```
^(?#start of the line)\d(?#digit).*(?#rest of the line)
```

.NET、Python、Ruby、PHP、Objective-C都支持这种记法，Java、JavaScript、Golang不支持。不过，还有一种注释的写法是各种语言都支持的，就是使用注释模式，此时，正则表达式对应的字符串可以跨越很多行。[1] 注释模式的代码见例5-5。

例5–5 注释模式

```
multilineString = "1 line \nNot digit\n2 line"
lineBeginWithDigitRegex = r"""
(?mx)     #enable multiline and extended mode
^         #start of the line
\d        #digit
.*        #rest of the line
$         #end of the line
"""
print re.findall(lineBeginWithDigitRegex, multilineString)
['1 line', '2 line']
```

在注释模式下，正则表达式内部的空白字符都被忽略（一般来说，主要是ASCII编码中的空白字符，Unicode编码中的空白字符情况不定），注释则以`#comment`的形式添加在正则表达式内部，每一条注释从`#`开始，到行末结束。许多文档中都用这种模式来解释复杂的表达式，并且会使用缩进表示层级结构，这样更加方便阅读和维护。比如匹配日期的正则表达式`((?x)(\d{4})-(\d{2})-(\d{2}))`，在注释模式下可以就这样像例5-6所示这样展开。

[1] 这里为了方便展示，使用了Python中的三引号字符串（Triple-Quoted String），这是Python中的特殊写法，以三个引号为首尾标志，字符串的值完全就是期间的文本“看上去的样子”，所以这个字符串横跨了多行。但是三引号字符串中也会进行字符串转义，所以在之前添加了`r`，表示原生字符串。

例 5–6　在注释模式下展开复杂正则表达式

```
dateRegex = r"""
(?x)                #enable multiline and extended mode
(                   #start of whoe regex
    (\d{4})         #year
    -               #dash
    (\d{2})         #month
    -               #dash
    (\d{2})         #day
)                   #end of whoe regex
"""
```

注释模式对应的模式修饰符是 x（extended mode，扩展模式，但更常见的写法是 free-spacing mode，**宽松格式模式**）。表 5-5 介绍了各语言中注释模式的预定义常量。

表 5–5　各语言中注释模式的预定义常量

语言	常量
.NET	`RegexOptions.IgnorePatternWhitespace`
Java	`Pattern.COMMENTS`
JavaScript	`/`*regex*`/x`
PHP	`/`*regex*`/x`
Python	`re.X` `re.VERBOSE`
Ruby	`Regexp::EXTENDED` `/`*regex*`/x`
Objective-C	`NSRegularExpressionAllowCommentsAndWhitespace`
Golang	暂不支持这种模式

你可能注意到了，例 5-6 同时指定了两种模式：多行模式和注释模式。注释模式的 x 与多行模式的模式修饰符 m，合写作`(?mx)`。如果需要同时使用多种模式，只要在`(?`*modifier*`)`中将模式修饰符排列起来就可以了。

如果希望同时指定多行模式和注释模式，使用预定义常量该怎么做？答案是，使用位运算符 |。通常来说，匹配模式对应的预定义常量都是 int 类型，所以多个值进行按位与的结果，并不会彼此干扰。比如在 Java 中，对应的写法就是 `Pattern.COMMENTS | Pattern.MULTILINE`，在.NET、Ruby 和 Python 中也可以这样。

如果是PHP和JavaScript，则在结束的分隔符之后直接并列模式对应的修饰符，比如`/`*regex*`/mx`，这种写法在 Ruby 中也行得通。

5.5 补充

5.5.1 更多的模式

上面提到的 4 种常用模式是各种语言中通用的，但是匹配模式并不只有这 4 种，不同的语言提供了不同的模式，它们一般都在预定义常量中，而且不一定有对应的模式修饰符。

比如 Java 的 `Pattern` 还包含其他模式（仅列举两种模式，更多的请参看文档）。

`Pattern.UNIX_LINES`：在此模式下，`^`、`.`、`$`识别的“行终止符”只有`\n`，而不能是其他字符（比如`\r\n`），它对应的模式修饰符是 `d`。

`Pattern.CANON_EQ`：在此模式下，字符的“相等”规则更加灵活，Unicode 字符 `a\u030A` 与`\u00E5` 也是“相等”的（都是 å，也就是拉丁字母 a 加上分音符，只是前者用两个字符组合，后者用单个字符），此模式可能对性能有很大影响，而且没有对应的模式修饰符。

Python 的 `re` 也包含了其他模式（仅列举两种模式，更多的请参阅文档）。

`re.U` 或 `re.UNICODE`：在此模式下，`\w`、`\d`、`\s` 等字符组简记法的匹配规则会发生改变，比如`\w` 能匹配 Unicode 中的“单词字符”，包括中文字符，`\d` 也能匹配 １、２之类的全角数字字符，它有对应的模式修饰符 `u`。

`re.A` 或 `re.ASCII`：因为在 Python 3 以上的版本中，正则表达式默认采用 Unicode 匹配规则，如果希望让`\d`、`\w` 等字符组简记法恢复到 ASCII 匹配规则，可以使用此模式，它有对应的模式修饰符 `a`。

在.NET 和 PHP 中，也有其他模式可用。不过一般来说，本章介绍的 4 种模式最为常用，在其他模式中，只有涉及 Unicode 或者 ASCII 的模式使用较多，因为它们影响了字符组简记法`\d`、`\s`、`\w` 的匹配规则。关于每种语言可用模式的具体信息，请参考该语言对应的章节或文档。

5.5.2 修饰符的作用范围

常见的模式修饰符是`(?modifier)`形式的，它表示“从现在开始使用某个模式”。通常的做法是将它写在正则表达式的最开头，表示“整个正则表达式都指定此模式”；如果它出现在正则表达式当中，则表示此模式从这里开始生效；[1]如果模式修饰符出现在某个括号内——比如`((?modifier)…)`——那么它的作用范围只限于括号内部，此模式也可以记为`(?modifier:…)`。

模式修饰符的作用范围见表 5-6 所示。

[1] Python 的情况有所不同，模式修饰符(?modifier)只要出现，无论在什么位置，都对整个正则表达式生效。

表 5–6　模式修饰符的作用范围

正则表达式	能匹配的文本
t(?i)he	tHE the thE tHe
th(?i)e	thE the
t((?i)h)e	tHe the
t(?i:h)e	tHe the

模式修饰符对正则表达式的操控性更强，因为预定义常量指定的匹配模式是对整个表达式生效的。

5.5.3　失效修饰符

`(?modifier)`指定了匹配模式开始作用的范围，在正则表达式中，另有一类失效修饰符，它用来“终止”某种模式的作用范围，其形式是`(?-modifier)`，类似`(?modifier)`，只是问号`?`之后多了一个减号`-`，表示“取消模式”，也就是**某个模式生效到此处为止**。

假设，一段文本中包含了许多单词，其中以 bar 结尾的有 foobar、zeebar 等，而且大小写不定。

```
Foobar feeBAR zeebAR ZEEBAR foobar zeeBAR
```

现在希望找出所有这样的 foobar 和 zeebar：不论 foo 和 zee 的大小写如何，只要结尾是大写的 BAR 就可以。FooBAR、ZEEBAR、zeeBAR 都符合要求，Foobar、zooBar、feeBAR 等则要排除。

你可能会觉得这很简单，要找到的单词，前面部分可能是 zee 或是 foo，不区分大小写则用字符组把大小写形式都列出来即可，后面必定是 BAR，所以正则表达式是`[fFzZ][oOeE][oOeE]BAR`。可是，这个表达式行不通，因为它可以匹配 feeBAR，而这并不是我们需要的。

真正要做的，其实是准确划定不区分大小写模式的作用范围即可：匹配前面部分的表达式是`(foo|zee)`，同时必须使用不区分大小写模式；匹配后面部分的`BAR`则必须停止使用不区分大小写模式。

要满足这个要求，可以使用上一节介绍的`((?i)foo|zee)BAR`，也可以使用失效修饰符，将表达式写作`(?i)(foo|zee)(?-i)BAR`，代码见例 5-7。因为 Python 不支持这种写法，所以使用了 Ruby（Ruby 的正则使用细节请参考第 15 章）。

例 5–7 不区分大小写模式与反向引用

```
str =" Foobar feeBAR zeebAR ZEEBAR foobar zeeBAR"
regex = /(?i)(foo|zee)(?-i)BAR/
puts str.scan(regex)
["feeBAR",  "ZEEBBAR",  "zeeBBAR"]
```

一般来说，只要语言或工具支持模式修饰符`(?modifier)`，都可以支持失效修饰符`(?-modifier)`，两者配合使用，可以更精确地设定模式的作用范围。不过，Python 和 JavaScript

并不支持(?-*modifier*)的写法。Golang 的文档里虽然没有明确指出支持，但实际代码测试可以支持。

5.5.4 模式与反向引用

反向引用前面介绍过，它引用之前的表达式匹配的文本。如果之前的表达式使用了某种模式，引用时是否会继承这种模式呢？下面看看实际情况，因为在 Python 中无法限定模式的作用范围，现在以 Java 为例（具体语法请参考第 11 章），代码见例 5-8。

例 5–8 不区分大小写模式与反向引用

```
//不区分大小写模式
"abcABC".matches("((?i)abc)\\1");    // => False
"abcABC".matches("(?i)(abc)\\1");    // => True
```

在表达式((?i)abc)\1中，\1引用的是(?i)abc能匹配的文本（在这个例子里就是abc），因为\1不在区分大小写模式的作用范围内，所以无法匹配ABC；而在表达式(?i)(abc)\1中，\1处在区分大小写模式的作用范围内，所以可以成功匹配ABC。

在例 5-9 所示的两个表达式中，\1引用的是之前a.匹配的文本，在单行模式下.可以匹配换行符；所以a.匹配的是字符a和换行符\n，因此\1也可以匹配字符a和换行符\n。

例 5–9 单行模式与反向引用

```
//单行模式
"a\na\n".matches("((?s)a.)\\1");     // => True
"a\na\n".matches("(?s)(a.)\\1");     // => True
```

在例 5-10 的两个表达式中，\1引用的是之前^a匹配的文本，^a只能匹配行开头的a；但是无论\1是否处在多行模式的作用范围内，它能匹配的不只有行开头的a，而是任何位置的字符a，这一点值得注意。

例 5–10 多行模式与反向引用

```
//多行模式
"aba".matches("((?m)^a).\\1");  // => True
"aba".matches("(?m)(^a).\\1");  // => True
```

5.5.5 冲突策略

使用(?-*modifier*)可以终止某个模式的作用范围；但是，模式也可以通过预定义常量来指定，它是对整个表达式生效的。这时候，就可能产生冲突。

如果在正则表达式中使用`(?-i)`取消不区分大小写的匹配模式，又使用了预定义常量（比如Java中的`Pattern.CASE_INSENSITIVE`）指定整个表达式采用不区分大小写的模式，这时候情况会是怎样呢？从例5-11可以看到（因为Python不支持失效修饰符，本节的例子均使用Java，Java的正则使用细节参见第11章）。

例5–11　冲突策略

```
Pattern.compile("(?-i)a",
    Pattern.CASE_INSENSITIVE).matches("A").find(); // => False
```

可以看到，**模式修饰符具有更高的优先级**。严格地说，如果用预定义常量指定了整个正则表达式采用某种模式，同时正则表达式内部使用了关闭该模式的失效修饰符，则失效修饰符之后的部分，都不受预定义常量所指定模式的影响（失效修饰符之前的部分则不受影响）。并不是每种语言的文档都会针对这种情况做详细讲解，但大家的原则一致。例5-12给出了具体的例子。

例5–12　更详细的例子

```
Pattern regex = Pattern.compile("a(?-i)a", Pattern.CASE_INSENSITIVE);
regex.matches("aa").find(); // => True
regex.matches("Aa").find(); // => True
regex.matches("aA").find(); // => False
```

5.5.6　哪种方式更好

模式修饰符和预定义常量都可以指定匹配模式，哪种方式更好？这个问题没有标准答案。

用模式修饰符指定的好处之一是简洁，因为通用的模式修饰符就只有`i x s m`，虽然不那么直观，但习惯之后并不难理解；而且，这些记法在各语言中都是通用的；再者，模式修饰符还可以用`(?i)`、`(?-i)`来精确控制模式的作用范围，这是预定义常量做不到的；另外，许多正则处理函数（详见第6章）只接受字符串形式的正则表达式，此时只能以模式修饰符来标识模式。

使用预定义常量的好处在于，它很形象——“单行模式”“多行模式”的说法确实很让人困惑，尤其它们竟然可以同时使用，互不干扰，而看字面意思分明是互相矛盾的；而且，一些模式并没有对应的修饰符，比如Java中的`Pattern.LITERAL`（消除所有元字符的特殊含义）、.NET中的`RegexOptions.ECMAScript`（字符组简记法采用ASCII匹配规则）。

我的建议是，熟练掌握常用的模式，**在能够使用模式修饰符的地方，尽量使用修饰符**。毕竟，无论使用哪种编程语言，`(?i)`是熟悉正则表达式的人都能看懂的，但是不是每个人都能记得的，Java中的`Pattern.DOTALL`等价于.NET中的`RegexOptions.Singleline`，也不是每个人都能记得，Java中的`Pattern.CASE_INSENSITIVE`等价于`RegexOptions.IgnoreCase`，因此，理解起来总是要多点周折。

第 6 章　其他

前 5 章已经介绍了正则表达式常用的功能和结构，但在实际应用中，经常可能遇到一些繁杂具体的问题，而常见的文档中往往不会涉及这类问题。所以有必要用一章的篇幅讲解实际中经常遇到的问题，澄清若干容易混淆的概念。

6.1　转义

正则表达式中的转义是一个麻烦的问题，经常会搞得人头疼。虽然之前各章都提到了转义，仍然有必要专门介绍。

6.1.1　字符串转义与正则转义

理解转义的基础是，明白字符串与正则表达式的关系。通常说的*string*（字符串）中，*string* 称为**字符串文字**（String Literal）[1]，它是某个字符串的值在源代码中的表现形式。比如字符串文字 `\n`，它包含`\`和`n`两个字符，意义（或者说它的值）是一个换行符（为方便观察，表示为NL）。在生成字符串时，应当进行“字符串转义”，才能准确识别字符串文字中`\n`的意思，如表 6-1 所示。

表 6-1　常见的字符串转义

字符串文字	字符串	说明
`\n`	NL	换行符
`\t`	Tab	制表符
`\\`	`\`	反斜线字符

另一方面，在源代码中看到的“正则表达式”*regex*，其中的 *regex* 称为**正则表达式文字**（Regular Expression Literal，以下简称正则文字），是正则表达式的表现形式。比如正则表达式`\d`，其正则文字包含`\`和 `d` 两个字符，它的意义（或者说值）是匹配数字字符的字符组简记法。在生成正则表达式时应当进行“正则转义”，才能将正则文字中的`\d` 识别为字符组简记法。

[1] 请参考 http://en.wikipedia.org/wiki/String_literal。

这两点都不难理解，但从字符串转到正则表达式的过程就比较复杂。因为不少正则表达式都是以字符串形式提供的，所以必须经过从字符串文字到正则表达式的转换。根据上面的介绍，字符串文字首先必须经过“字符串转义”，才能得到真正的字符串；接下来，这个字符串作为正则文字，经过“正则转义”，最终得到正则表达式。图 6-1 说明了从字符串文字到正则表达式的转义过程，表 6-2 则列出了若干常用字符串的转义。

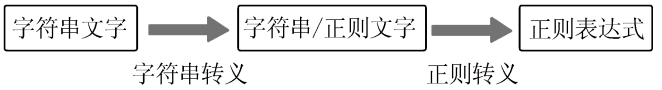


图 6–1　字符串转义与正则转义

表 6-2 以常见的几个字符为例，详细说明了这种转换过程。

表 6–2　从字符串文字到正则表达式的转换

字符串文字	字符串/正则文字	正则表达式	说明
\\n	\n	NL	换行符
\\t	\t	Tab	制表符
\\\\	\\	\	反斜线字符

换行符和制表符的情况比较容易理解，它们的正则文字是\n 和\t，也就是字符串的值。但是从字符串文字必须经过字符串转义才能得到作为正则文字的\n 和\t，根据字符串转义的规则，反斜线字符\必须写成\\，所以字符串文字必须写成\\n 和\\t。

如果要表示正则表达式中的\，必须使用正则文字\\，这样在正则转义时才能正确识别。同时，正则文字中的每个\都必须用字符串文字\\表示，所以正则表达式\对应的字符串文字就是\\\\。网络上常有人问，某个正则表达式中为什么需要连续用到 4 个反斜线字符，原因就在这里。

例 6–1　\的转义

```
#\\\\经过字符串转义和正则转义，得到的正则表达式就是一个\
#\\经过字符串转义，得到的字符串就是一个\
re.search("\\\\", "\\") != None      #  => True
```

有时候情况更复杂：正则表达式中的换行符或者制表符，在字符串文字中必须写成\\n 或\\t；但是，用\n 或\t 也没有问题。原因在于，在处理字符串转义时，它们已经被解释为换行符或制表符，所以传递给正则表达式的字符串中就包含了换行符或者制表符，具体情况如表 6-3 所示。

表 6–3 从字符串文字到正则表达式的转换

字符串文字	字符串/正则文字	正则表达式	说明
\\n	\n	NL	换行符
\n	NL	NL	换行符
\\t	\t	Tab	制表符
\t	Tab	Tab	制表符

从例 6-2 可以看到，字符串文字中的\\n 和\n 表示的是同一个字符，只是前者先做字符串转义，再做正则转义，而后者只做了字符串转义。看来，似乎还是\n 更符合直觉一些——如果你确实能够理解，为什么字符串文字中\\n 和\n 的结果竟然一样，使用\n 确实可行。但是，正则表达式中\的字符串文字还是必须写成\\\\。

例 6–2 \n 和\t 的转义

```
re.search("\\n", "\n") != None   #  => True
re.search("\n", "\n") != None    #  => True
re.search("\\t", "\t") != None   #  => True
re.search("\t", "\t") != None    #  => True
```

要特别注意的是\b。在一般字符串中，\b 是预定义的转义序列，表示退格符（backspace，表示为 BS）；但是在正则表达式中，它表示单词边界（记为\B）。如果在字符串文字中写\b，字符串转义为退格符，作为正则表达式交给正则表达式，正则表达式真正得到的就是退格符 BS，而不是单词边界\B，如表 6-4 和例 6-3 所示。所以，如果用到了单词边界，在字符串文字中一定要写成\\b。所以，最保险的办法是：正则表达式中的每一个\，在字符串文字中都要写成\\。本书之前章节中的代码，一直遵循的是这样的规则。

表 6–4 \b 的转义

字符串文字	字符串/正则文字	正则表达式	说明
\b	BS	BS	退格符
\\b	\b	\B	单词边界

例 6–3 \b 的转义

```
re.search("\ba\b", "a") != None      #  => False
re.search("\\ba\\b", "a") != None    #  => True
```

在 Python 中也可以这样，如例 6-4 所示。

例 6–4 Python 中 \d 的转义

```
re.search("\d", "1") != None     #  => True
re.search("\(", "(") != None     #  => True
```

看起来足够奇怪，因为在字符串中\d 和\(都不是合法的转义序列，所以如果在 Java 中这样写，是会报编译错误的。为什么 Python 中可以呢？这是因为 Python 对字符串有特殊的规定：如果遇到无法识别的转义序列，则将它**原封不动**地保存下来，具体的处理见表 6-5。

表 6–5　\d 的转义

语言	字符串文字	字符串/正则文字	正则表达式	说明
Python	\\d	\d	\d	字符组简记法
	\d	无法识别，原样保存	\d	字符组简记法
Java	\\d	\d	\d	字符组简记法
	\d	无法识别，报错	×	×

在 PHP 中也有同样的规定，所以在 PHP 的字符串文字中也可以直接写\d、\w、\s 等，效果和\\d、\\w、\\s 是一样的。

这样看来，使用字符串形式的正则表达式，转义的处理确实比较复杂。最好是能省去这些麻烦——正则表达式是怎样的，正则文字中就怎样写。要做到这一点，有两种办法：第一，使用**原生字符串**（Raw String），也就是完全忽略字符串转义的特殊字符串，在 Python、Ruby、.NET、Golang 中都提供了原生字符串。

Python

```
#在字符串之前用 r 标注原生字符串
pattern = re.compile(r"a\.\nb\b")
```

Ruby

```
#Ruby 中用单引号标注原生字符串
pattern = new Regexp('a\.\nb\b')
```

C#

```
//C#中用@标注
Regex regex = new Regex(@"a\.\nb\b");
```

VB.NET

```
//VB.NET 中的字符串与其他语言的不一样，只有双引号字符"需要转义
Dim regex as new Regex("a\.\nb\b");
```

Golang

```
//Golang 中的原生字符串用反引号`（不是单引号，是键盘上 1 左边那个键）
reg := regexp.MustCompile(`a\.\nb\b`)
```

第二，直接使用正则文字，在 Ruby 和 JavaScript 中可以这么做。下面以正则表达式 a\.\nb\b 为例。这种方法一般是直接在正则表达式两端添加分隔符/。此时，字符串转义会被略过，不过另外还有一个字符需要转义，就是分隔符/。

Ruby

```
pattern = Regexp.new(/a\.\nb\b/)
```

```
#下面这个表达式只包含一个字符，就是/
pattern = Regexp.new(/\//)
```

JavaScript

```
var pattern = new RegExp(/a\.\nb\b/);
//下面这个表达式只包含一个字符，就是/
var pattern = new RegExp(/\//);
```

总的来看，转义的情况确实很复杂。我推荐的做法是：

第一，如果可以使用原生字符串或者正则文字直接表示正则表达式，应当尽量使用这种做法，因为它简单直观，方便理解；[1] 而且，本书从此开始，如果能使用原生字符串表示正则表达式，则使用原生字符串，在Python示例代码中使用`r"regex"`的形式。

第二，如果确定必须使用字符串文字，请尽量坚持这条原则：正则表达式中的每个反斜线`\`，在字符串文字中都必须写成`\\`，只有`\n`、`\t`中的反斜线例外（但是`\\n`和`\\t`也不难理解）；我**不推荐**使用语言自身提供的“字符串转义时将无法识别的转义序列保留下来”的规定——比如要在正则表达式中使用字符组简记法`\d`，正则文字中要写为`\\d`，而不是`\d`——这样有利于使用其他语言的程序员阅读。

6.1.2 元字符的转义

了解了字符串与表达式的关系，再来看正则表达式中元字符的转义。元字符是有特殊含义的字符，如果要匹配“元字符”自身则必须转义，也就是在元字符之前添加反斜线`\`。比如元字符点号`.`，可以匹配除换行符以外的任何字符，如果要准确匹配字符串中的点号`.`，正则表达式中就必须写`\.`。

也有些时候，匹配元字符自身并非一定要转义，表 6-6 列出了各种结构的转义。

表 6–6 常用结构的转义

结构	记法	转义	说明
字符组	`[]`	`\[]`	只对开方括号转义
	`.`	`\.`	
	`-`	`\-`	`[a\-b]`等价于`[-ab]`
量词	`* + ?`	`\* \+ \?`	
	`*? +? ??`	`\*\? \+\? \?\?`	
	`{m,n}`	`\{m,n}`	只对开花括号转义
括号	`(…)`	`\(…\)`	开、闭括号都需要转义

[1] 考虑到不是所有语言都支持原生字符串，所以本书的例子中，一般还是通过字符串来生成正则表达式，以利于读者理解，虽然有时候这样做确实很麻烦。

（续表）

结构	记法	转义	说明
多选结构	|	|	竖线需要转义
括号和多选结构	(…|…)	\(…|…\)	开、闭括号和竖线都要转义
断言	^ $	\^ \$	
替换引用	$num	\$或$$	在替换的 *replacement* 字符串中转义

从表中可以看到，对称出现的元字符在转义时并不是“对称”的，比如与开方括号[对应的闭方括号]，与开花括号{对应的闭花括号}，这两个字符是否是元字符，取决于**之前是否出现**了开方括号或开花括号。如果出现了，则作为元字符出现；否则，作为普通字符出现。表 6-7 以方括号为例，说明了这一点。

表 6–7　字符组只要求转义开方括号

正则表达式	解释	能匹配的字符串
[ab]	字符组[ab]	a b
\[ab]	字符[、字符 a、字符 b、字符]	[ab]
ab]	字符 a、字符 b、字符]	ab]

字符组内部的闭方括号]在**任何情况下都要转义**，否则类似[]]的正则表达式会出现二义性，造成识别错误。所以能匹配字符 a、字符 b、字符]的字符组，应当写为[ab\]]，而不是[ab]]。同样道理，括号内部的任何闭括号)都要转义，比如包含 ab 和 b)的多选结构的正则表达式就应当写为(ab|c\))，而不是(ab|c))。

另一点容易忽略的是，在进行正则表达式替换时，*replacement* 字符串中也可能出现转义。比如在 Java 中，*replacement* 里通过*$num* 引用对应的捕获分组。如果想在替换时使用一个单独的$符号，而不是引用分组（比如生成价格字符串），则会报错，必须做转义才可以解决问题，在 Java 中使用\$，代码见例 6-5。

例 6–5　Java 中$的转义

```
System.out.println("2010-12-20".replaceAll("(\\d{4})-(\\d{2})-(\\d{2})", "$2/$3/$1"));
12/20/2010

System.out.println("the price is 12.99".replaceAll("\\d+\\.\\d{0,2}", "$$0"));
Exception in thread "main" java.util.regex.PatternSyntaxException: Illegal
group reference

//转义需要出现的$
```

```
System.out.println("the price is 12.99".replaceAll("\\d+\\.\\d{0,2}", "\\$$0"));
the price is $12.99
```

.NET 中的 *replacement* 中同样使用*$num* 引用对应的捕获分组，但是.NET 中$的转义并不是\$，而是$$，代码见例 6-6。

例 6–6 .NET 中$的转义

```
Console.Write(Regex.Replace("the price is 12.99", "\\d+\\.\\d{0,2}", "$$$0"));
the price is $12.99
```

6.1.3 彻底消除元字符的特殊含义

有些时候需要消除所有元字符的特殊含义，全部作为普通字符。这种情况经常发生在处理用户输入的场合：用户输入了某个字符串，需要根据这个字符串进行正则表达式查找。

假设某个文件中包含多个文本片段，用空格分隔，现在需要查找包含用户输入内容的片段：用户输入 cat，查找包含 cat 的行，直接的思路是^.*cat.*$（使用多行模式，同时不能指定单行模式），假设用户输入的内容保存在变量 *userInput* 中，就应当用*"^.*" + userInput + ".*$"* 得到查找用的正则表达式。

如果用户输入 cat，这么做当然没有问题。如果用户输入的是 ca*t，得到的正则表达式就是^.*ca*t.*$：本意是查找包含字符串 ca*t 的行，但*是正则表达式中的元字符，正则表达式ca*t 能匹配字符串 cat、caat、caaat，却不能匹配 ca*t，这样就会产生错误。更麻烦的是，恶意用户可能会输入 a(b*b*)b*之类的字符串，这样的正则表达式匹配起来会消耗非常多的资源，这就是**正则表达式拒绝服务攻击**（☞ 143），严重时可能把服务程序打垮。

为解决这类问题，一些语言中的正则表达式提供了特殊结构，彻底消除元字符的特殊含义，提供真正“安全”的表达式（也就是普通字符串）。最常见的做法是在需要消除元字符特殊含义的正则表达式两端添加\Q 和\E，也就是\Q…\E（其中 Q 表示“引用文本 Quoted”，而 E 表示“引用文本结束 End”），像例 6-7 那样。此时正则表达式\Qca*t\E 就不再能匹配 cat、caat、caaat，只能匹配 ca*t。\Q…\E 只对在它内部的子表达式生效，所以在正则表达式^.*\Qca*t\E.*$中，^.$和.*$会按照正则表达式的默认规则匹配，只有中间的 ca*t 作为普通字符串匹配，整个表达式能够匹配的就是确定包含 ca*t 的行。

例 6–7 用\Q…\E 消除元字符的特殊含义

```
//因为 Python 不支持\Q…\E 的表示法，现在以 Java 语言举例
//没有使用\Q…\E
"cat".matches("ca*t");      // => True
"caat".matches("ca*t");     // => True
```

```
"caaat".matches("ca*t");    // => True
"ca*t".matches("ca*t");     // => False
//使用\Q…\E，注意在正则文字中必须双写反斜线为\\Q…\\E
"cat".matches("\\Qca*t\\E");      // => False
"caat".matches("\\Qca*t\\E ");    // => False
"caaat".matches("\\Qca*t\\E ");   // => False
"ca*t".matches("\\Qca*t\\E ");    // => True
//对正则表达式中的某个子表达式使用\Q…\E
"regex like ca*t and so on".matches("^.*\\Qca*t\\E.*$");   // => True
"has one cat".matches("^.*\\Qca*t\\E.*$");    // => False
```

`\Q…\E` 并不是所有语言都支持的，在本书介绍的语言中，明确支持它的有 Java、PHP、Objective-C、Golang。其他语言虽然不支持`\Q…\E`的记法，但其中一些也提供了专门的处理函数消除元字符的特殊含义；比如.NET 的 `Regex.Escape(`*`text`*`)`，它接收一个字符串，返回的字符串中的所有元字符都做了转义处理，消除了特殊含义，从其结果字符串生成的正则表达式，就只能匹配与 *`text`* 完全一样的字符串，代码见例 6-8。

例 6–8　用专门的函数消除元字符的特殊含义

```
Regex.IsMatch("ca*t", Regex.Escape("ca*t"));    // => True
Regex.IsMatch("(ab)", Regex.Escape("(ab)"));    // => True
```

表 6-8 列出了各种语言中消除元字符特殊含义的函数。

表 6–8　常用语言中消除元字符特殊含义的函数

语言	函数
.NET	`Regex.Escape(`*text*`)`
Java	`Pattern.quote(`*text*`)`
PHP	`preg_quote(`*text*`)`
Python	`re.escape(`*text*`)`
Ruby	`Regexp.quote(`*text*`)` `Regexp.escape(`*text*`)`
Objective-C	`(NSString *)escapedPatternForString:(NSString *)`*text*
Golang	`Regexp.QuoteMeta(`*text*`)`

注：PHP 中的正则表达式两端必须出现分隔符，所以 preg_quote()可以设定第二个参数明确指定分隔符；否则，假如分隔符是/，而 text 又包含/，就可能发生冲突。

请记住，使用正则表达式时，仔细分辨并消除用户输入字符串中元字符的特殊含义，是不可忽略的步骤。

6.1.4 字符组中的转义

在正则表达式中，如果需要表示作为元字符的普通字符（比如*、?、(等），就需要使用转义，这一点不存在疑义。特殊的是，常见的元字符出现在字符组内部基本都不算元字符，也就是说，它们在字符组内部出现时，不需要转义。代码如例 6-9 所示。

例 6–9 字符组内部几乎没有元字符

```
re.search(r"[*]","*") != None    #  => True
re.search(r"[?]","?") != None    #  => True
re.search(r"[()","(") != None    #  => True
```

之前讲过，字符组有自己的元字符规定，也有相应的转义规定：在字符组内部，只有三个字符需要转义。一个是闭方括号]，如果不是作为字符组结束标志的闭方括号，则必须写成\]，比如[0\]9]，它可以匹配的字符是 0、]、9；一个是横线-，如果不是用于范围表示法（比如[0-9]），必须写成\-，比如[0\-9]，它可以匹配的字符是 0、-、9，当然如果它紧跟在开方括号之后，也可以不用转义，[0\-9]和[-09]是等价的（我更推荐后一种写法，因为更清晰简洁）；还有一个需要转义的字符是^，如果它不是用于排除型字符组（[^ab]），则应当写成\^，比如[\^ab]，它可以匹配除^、a、b 之外的任何字符，如果它不是紧跟在开方括号之后，也不用转义，[\^ab]、[a^b]、[ab^]是完全等价的（我更推荐后两种写法，因为更清晰简洁）。从例 6-10 可以看到，这两种写法的结果是相同的。

例 6–10 字符组中三个需要转义的元字符

```
#转义的]
re.search(r"[0\]9]","]") != None     #  => True
#未转义的-
re.search(r"[0-9]","-") != None      #  => False
#转义的-
re.search(r"[0\-9]","-") != None     #  => True
re.search(r"[-09]","-") != None      #  => True
#未转义的^
re.search(r"[^ab]","^") != None      #  => True
re.search(r"[^ab]","a") != None      #  => False
#转义的^
re.search(r"[\^ab]","^") != None     #  => True
re.search(r"[\^ab]","a") != None     #  => True
re.search(r"[a^b]","^") != None      #  => True
re.search(r"[a^b]","a") != None      #  => True
```

6.2 正则表达式的处理形式

在之前讲解正则表达式时主要使用的 Python 语言中，其具体办法是调用 re 这个 package 中的

函数（方法），比如 re.*search*()、re.*findall*()、re.*sub*()。选择合适的函数，将正则表达式和字符串传入，这是一种常见的方式，比如 PHP 也是如此。但是，正则表达式并不只有这一种处理形式，比如 Java 语言的处理形式就与 Python 和 PHP 大不一样，所以下面简单介绍常见的处理形式。

6.2.1　函数式处理

在函数式处理中，正则表达式的常见操作（查找、替换等）都有对应的函数，执行这些操作时，调用对应的函数，将正则表达式和字符串作为参数传入即可，Python、PHP 是函数式处理的典型代表。

Python

```
#查找
re.findall(r"\d+","123 45 6")
#替换
re.sub(r"\d+", r"[\g<0>]"," 123 45 6")
```

PHP

```
//查找
preg_match("/\\d+/","123 45 6");
//替换
preg_replace("\\d+", "[$0]"," 123 45 6");
```

6.2.2　面向对象式处理

Java、.NET、Objective-C、Golang 之类的语言中的正则表达式采取了不同的处理形式：进行正则表达式处理之前，必须生成专门的正则表达式对象（在不同的语言里，对象所属的类名不同），再调用此对象的成员函数。

Java

```
//必须先生成 Pattern 对象，再生成 Matcher 对象
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matches("123 45 6");
while(matcher.find()) {
    //do as you wish
}
```

.NET

```
//必须先生成 Regex 对象，再生成 Match 对象
Regex regex = new Regex(@"\d+");
Match match = regex.Match("123 45 6");
while (match.Success) {
    //do as you wish
    match = match.NextMatch();
}
```

6.2.3 比较

同样是进行查找操作，如果使用函数式处理，只需要调用对应的函数；但使用面向对象式处理，需要分步进行，首先生成各种对象，再调用对象自身的方法。看起来后者要麻烦很多，这种处理方式有什么好处呢？

之前讲过，正则表达式并不等于字符串，即便正则表达式是以字符串形式给出的，进行正则表达式操作之前，必须首先生成专用的“正则表达式对象”。函数式处理隐去了生成的过程，感觉更加直接；面向对象式处理则暴露了生成的过程，感觉更加细致。

只要执行正则表达式操作，就会产生正则表达式对象；所以，面向对象式处理的代码虽然更烦琐，但是如果需要重复用到某些表达式，它的效率往往远高于函数式处理——在面向对象式处理中，可以将已经生成的对象作为变量保存起来，就不必重复生成了。

面向对象式处理的步骤分明、效率更高，而函数式处理的好处是方便顺手、代码简洁。所以最好的办法是，**根据应用场合的不同，采取不同的处理方式**：如果正则表达式只是单次使用，则选择函数式处理；如果正则表达式需要重复使用，则选择面向对象式处理。实际上，许多编程语言也提供了对应的设计，比如 Python 提供了一些面向对象式处理的方法，Java 和.NET 也提供了一些函数式处理的方法。

在 Python 中进行面向对象式处理，可以先调用 `re.compile()`生成专门的 `RegexObject` 对象（也就是正则表达式对象），在进行正则表达式操作时，可以把这个对象作为参数，传给对应的函数；另一方面，`re` 中各个函数也可作为对象自身的成员方法，所以调用生成好的 `RegexObject` 对象的成员方法，也是可以的，如例 6-11 所示。

例 6–11　Python 中可以使用两种处理方式

```
#函数式
re.findall(r"\d+","123 45 6")
#面向对象式
#生成 re 对象
digitRegex = re.compile(r"\d+")
#调用 re.findall 方法
re.findall(digitRegex, "123 45 6")
#或直接调用 re 对象的方法
digitRegex.findall("123 45 6")
```

Java 语言虽然采用面向对象式处理，但也提供了一些便捷函数，比如 `Pattern.matches(regex, input)`，它用来验证 *input* 能否由 *regex* 匹配，如例 6-12 所示。

例 6–12　Java 中也可以使用函数式处理

```
Pattern.matches("\\d+", "123 45 6");
```

```
//它等价于
Pattern.compile("\\d+").matcher("123 45 6").matches();
```

.NET 中也提供了类似的方法，比如 `Regex.IsMatch(regex, input)`，它也用来验证 *input* 能否由 *regex* 匹配，代码如例 6-13 所示。

例 6–13　.NET(C#)中也可以使用函数式处理

```
Regex.IsMatch(@"\d+", "123 45 6");
//它等价于
(new Regex(@"\d+").Match("123 45 6")).Success;
```

6.2.4　线程安全性

如果采用面向对象式处理，可以将生成的正则表达式对象存储在变量中反复使用，以提高效率。在单线程编程的环境下，“反复使用”当然没有问题，如果多个线程共用同一个正则表达式对象，结果会如何呢？

仔细观察面向对象式处理可以发现，在面向对象式处理中，一般会出现两个对象：一个是单独对应正则表达式的，可以叫作“正则表达式对象”，比如Java中的`Pattern`、.NET中的`Regex`、Python中的`RegexObject`、Objective-C中的`NSRegularExpression`、Golang中的`Regexp`；另一个是在正则表达式与希望处理的字符串联系起来时生成的对象，可以叫作“匹配结果对象”，比如Java中的`Matcher`、.NET中的`Match`、Python中的`MatchObject`。[1] 这几个对象之间的关系如图 6-2 所示。

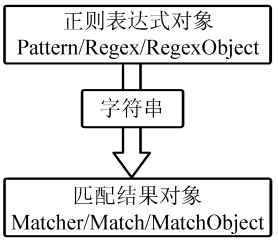


图 6–2　正则表达式操作中对象间的关系

正则表达式对象本身基本没有什么状态可言，所以**这个对象总是线程安全的**，可以由多个线程共享。匹配结果对象则需要维护自身状态，比如当前匹配是否成功，当前匹配的开始或结束偏移值等。如果多个线程共享同一个匹配结果对象，有可能遇到这样的情况：某个线程正在查询当前匹配的开始位置，另一个线程调用了 `matcher.find()`（Java 中）或者 `Match.nextMatch()`（.NET 中）之类的方法，尝试进行下一次匹配，就可能产生混乱。所以**匹配结果对象一般不是线程安全**

[1] 也有例外，比如 JavaScript 中的 RegExp，它既是正则表达式对象，其中又保存了与具体匹配相关的信息。不过 JavaScript 一般不会使用多线程编程，所以这个例外不做详细介绍。

的，也不应由多个线程共享。

因此，最理想的办法是：多个线程可以共享同一个正则表达式对象，节省时间；但操作不同文本时，应当针对各个线程生成专属的匹配结果对象，如图 6-3 所示。

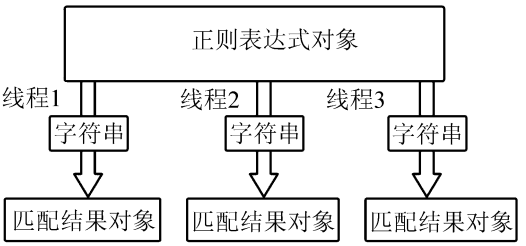


图 6–3 多线程环境下正确的处理模式

例 6–14 多线程环境下错误使用正则表达式（以 Java 为例）

```
//多个线程共享同一个 Matcher
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matches("123 45 6");
…
//thread 1
while(matcher.find()) {
     //do as you wish
}
…
//thread 2
matcher.reset("78 9 0");
while(matcher.find()) {
     //do as you wish
}
```

例 6–15 多线程环境下正确使用正则表达式（以 Java 为例）

```
//各个线程共享同一个 Pattern
Pattern pattern = Pattern.compile("\\d+");
…
//thread 1
Matcher matcher = pattern.matches("123 45 6");
while(matcher.find()) {
     //do as you wish
}
…
```

```
//thread 2
Matcher matcher = pattern.matches("78 9 0");
while(matcher.find()) {
    //do as you wish
}
```

6.3　表达式中的优先级

正则表达式千变万化，都是由之前介绍的字符组、括号、量词等基本结构组合而成的，只要掌握了组合的规则，面对再复杂的表达式，都能把结构梳理清楚。

仔细观察会发现，正则表达式的元素之间的组合关系只有 4 种。

普通拼接	abc
括号	(abc)
量词	a*b
多选结构	ab|cd

注：“普通拼接”可能是最常见的组合关系，只是平时不一定明确意识到，正则表达式 abc 就是 a、b 和 c 的普通拼接。a(bc)则可以看作 a 和(bc)的拼接。

除列出的 4 种组合关系外，正则表达式中的其他结构，比如环视(?=…)，都可以视为单独的元素，其中的正则表达式最终可以归类到上面的 4 种组合关系中。

所以，真正需要关心的就是这 4 种组合的优先级，详见表 6-9。

表 6–9　正则表达式中的优先级

优先级	组合	说明
1	(*regex*)	整个括号内的子表达式成为单个元素
2	* ? +	限定之前紧邻的元素
3	abc	普通拼接，元素相继出现
4	a|bc	多选结构

注：数字越小，优先级越高。

有了表 6-9，就可以拆解各种表达式了，表 6-10 给出了几个简单的例子。

表 6-10 正则表达式中的优先级举例

正则表达式	能匹配的字符串	说明
ab	ab	普通拼接
ab+	ab abb abbb	量词的优先级高于普通拼接
(ab)+	abc aabc aabc	括号的优先级高于量词
ab|cd	ab cd	多选结构的优先级低于量词
(ab|c)d	abd cd	括号的优先级高于多选结构和量词

如果你觉得上面这几个例子都不难理解，来看一个更复杂的表达式 `ab*(cd|e+)?|fg`，它的优先级划分关系如图 6-4 所示。

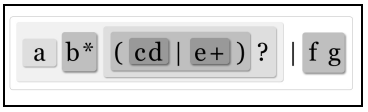


图 6-4 表达式的优先级

注：颜色越深，优先级越高。

判断优先级并不困难，一般来说，根据上面介绍的知识逐步拆解即可，但是多选结构值得单独强调。

常见的多选结构是以竖线配合括号`(option1|option2)`的形式给出来的；但是，多选结构并非必须与括号连用，如果没有括号，单独出现竖线`|`，也可以实现多选结构的功能，比如正则表达式 `ab|cd`，它能匹配的字符串和`(ab|cd)`是完全一样的。

比较起来，后一种写法更加清楚，因为两端的括号明确限定了竖线所标识的多选结构的范围，所以很难混淆`^(ab|cd)$`和`^ab|cd$`，后者等于`(^ab|cd$)`；而且在匹配之后，还可以使用分组及反向引用等功能（`\1`、`\2`之类），明确得到匹配的文本（不用括号则做不到这点），再进行下一步操作。但是，分组又会降低效率，尤其是在正则表达式非常复杂，或者要处理的文本非常多时，使用分组可能会严重影响效率。

考虑到这些因素，我推荐的做法是：应用多选结构一定不能省略括号，但同时，为了避免分组影响效率，如果不需要提取捕获文本，应当把普通的括号`(…)`改为非捕获型括号`(?:…)`，这样既明确标识了多选结构，又免去了分组捕获的成本。不过在本书中，为了格式简单清晰考虑，除非特别说明，一般不使用非捕获型括号。

6.4 回车和换行

许多程序员经常搞不清楚回车`\r` 和换行`\n` 到底有什么区别，只知道两个字符写法不同。然

而键盘上只有回车键，它起到的作用却是“换行”。在大多数时候，我们只是在编辑文本时敲一下回车键，然后本行结束，光标自动转到下一行的开始位置。因为平台和软件的不同，有时候输入的是换行\n，有时候输入的是回车换行\r\n，历史久远的某些平台可能会输入\r。为什么会这样？其中的区别到底在哪里？

其实，这是一个历史问题。

在文字处理软件还没有发明的年代，大家只能用机械打字机来打字，如今还可以在一些电影里看到它的样子。不同于今天计算机中的文字处理软件，可以由计算机来排版，机械打字机的每个键“打”在纸上的位置都是相同的，为了避免字符输入重叠，每打印一个字符，打印纸就向左移动一格，这样下一个字符才会打在当前字符的右侧。实现这个功能的装置的中文名是“打印头”，英文名是 carriage——大概因为它比较像马车吧。

在使用机械打字机时，每打印完一行，就必须敲“回车”，让打印头回到初始位置，再次打字仍然从最左侧开始。同时还不要忘记敲“换行”，让打印纸向上移动一行，否则两行就会重叠在一起，后一行的字符直接打在前一行已有的字符上。

今天在计算机中，无论回车还是换行，都不会让两行文字重叠在一起，所以把回车和换行分开显得有点怪异。不过讲清楚它们背后的历史，确实能解开许多人的疑惑。

第二部分

第 7 章　Unicode

虽然本书讲解的主题是正则表达式，而且 Unicode 已经不是什么新鲜概念，但根据我的经验，适当了解 Unicode 相关知识，绝对有助于更好地使用正则表达式，也有助于更好地处理文本。本章会简明扼要地介绍 Unicode 的基础知识，如果希望在处理文本时知其然而且知其所以然，阅读本章是不错的选择。

7.1　基础知识

在 Unicode 出现之前，各种系统的文字编码是各不相同的，比如中国大陆使用的是 GB2312，中国台湾使用的是 Big5，美国使用的是 ASCII（ISO-8859-1），日本使用的是 Shift JIS 等。这些编码系统的作用，都是给每个字符分配一个数字（码值）作为编码，然后进行码值-字符和字符-码值的双向映射操作。比如在 ASCII 编码中，字母 C 的编码是（十进制的）67，又比如在 GB2312 编码中，汉字“正”的编码是（十进制的）54781。

这样的标准在本地区使用当然没有问题，但如果要跨地区交流就会遇到困难，因为同样的码值，在不同的编码体系下对应的字是不同的。如果同一篇文档里要显示不同编码的文字，那更是不可能完成的任务。Unicode 的发明，就是为了解决这些问题。

Unicode 由两大部分组成，一部分是 UCS，也就是 Universal Character Set，它定义了一个广阔的空间，能把各种不同语言中的字符全部装进去，为每个字符分配唯一的码值（Unicode 中的术语是 Code Point）。另一部分是 UTF，也就是 Unicode Transformation Format，它定义了 UCS 中的每个码值以什么样的方式来传输（和存储）。

我们通常谈论的“Unicode 编码”，严格来说是指 UCS，也可以叫“字符集”，它关心每个字符对应的码值是多少，比如中文字符“正”的码值是十进制的 27491，十六进制的表示是 `6B63`。讨论 Unicode 时，常用的表示法是 `U+6B63`。在本章，讨论十六进制表示的码值都会采用 `U+6B63` 这样的写法，讨论十进制表示的码值时使用 27491 这样的写法。

码值只是概念，要变成字节才能够读取、存储、传输，所以要用到 UTF。如果采用如今常用的 UTF-8 编码，它需要用 3 字节：`E6 AD A3`（Web 相关开发里经常会需要对参数进行 URL Encode，如果你留意观察，会发现“正”字进行 URL Encode 之后就是`%E6%AD%A3`）。其他 UTF 还有 UTF-16、UTF-32、UCS-2、UCS-4 等，它们之间的关系如图 7-1 所示。

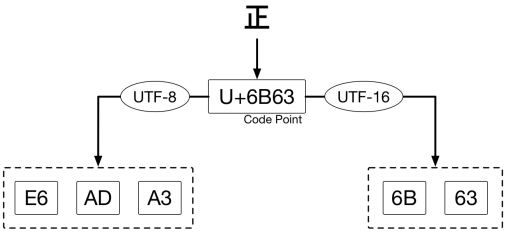


图 7–1 UCS、Code Point、UTF–8、UTF–16 的关系

为了能够在全世界通行，UCS 空间设计得很大，码值从 0 到 1 114 111，一共可以容纳超过 100 万个字符。按照 2017 年 6 月最新发布的 Unicode 10.0 标准，280 016（25%）已经分配完毕，其中 136 755（12%）分配给了字符，137 468（12.3%）分配给了私有用途，2048 分配给了代用码（surrogates），66 分配给了非字符用途。剩下 834 096（75%）尚未分配。

整个 UCS 又可以分为 17 个“平面”（Planes），每个平面包含 65 536 个码值（二者的乘积正好是 1 114 112）。这 17 个平面的编号从 0 到 16，其中编号为 0 的平面称为“基本多语言平面（Basic Multilingual Plane, BMP）”，对应的码值从 0 到 65 535，日常用到的绝大多数文字字符都包含在这个平面里。除了 BMP 之外，其他平面称为“补充平面（Supplementary Plane）”。

因为日常用到的字符基本都包含在 BMP 中，而 BMP 的空间大小是 65 536，是 2 的 16 次方，正好可以用 2 字节对应，所以早年的许多设计就以此为基础，后来产生了很多麻烦。

因为假设“所有的字符都可以用 2 字节对应”，所以早期出现的一种 Unicode 编码格式是 UCS-2，它用 2 字节表示所有字符。要注意的是，它的名字里虽然有 UCS，其实和 UTF-8 一样，是一种传输编码。最开始，UCS-2 确实工作得不错，但随着技术的发展，大家很快发现 BMP 不够用了，需要启用新的平面，而 UCS-2 因为先天限制，不可能有空间表示更多的字符了。于是，UTF-16 诞生了。

UTF-16 的设计思想非常巧妙。如果要表示的字符在 BMP 内，直接照搬 UCS-2 就可以。如果不在 BMP 内，也就是码值超过了 65 536，则利用 BMP 中预留的 `U+D800` 到 `U+DFFF` 这 2048 个代用码（本身是没有意义的）来解决。这 2048 个代用码分为两段，`U+D800` 到 `U+DBFF` 和 `U+DC00` 到 `U+DFFF`。对任何一个码值，只要其第一个字节在 `D8` 到 `DF` 之间，它一定不是 BMP 中原有的字符，所以可以放心用来当替代品表示其他字符。对于 BMP 之外的字符，UTF-16 定义的转换规则如下：

第一步，将码值减去 `0x010000`，得到一个 20 位的数值。

第二步，将这个数值的高 10 位加上 `0xD800`，得到的码值作为高位代用码（High Surrogate）。

第三步，将这个数值的低 10 位加上 `0xDC00`，得到的码值作为低位代用码（Low Surrogate）。

第四步，将高低两段代用码组合起来，得到 4 字节序列，就是该字符的 UTF-16 编码值。

UTF-16 转换举例见表 7-1。

表 7–1　UTF–16 转换举例

码值	二进制码值	UTF–16 二进制编码值	UTF–16 编码值
U+0024	0000 0000 0010 0100	0000 0000 0010 0100	0024
U+20AC	0010 0000 1010 1100	0010 0000 1010 1100	20AC
U+10437	0001 0000 0100 0011 0111	1101 1000 0000 0001 1101 1100 0011 0111	D801 DC37

也就是说，UCS-2 是 UTF-16 的子集，在 UTF-16 编码格式下，用于表示一个字符的字节数可能是变化的，或者是 2 个字符，或者是 4 个字符。

看起来 UTF-16 确实解决了空间不够的问题，但它也有问题：本来在 ASCII 编码中，英文字符只占用 1 字节，使用 UTF-16 之后则必须占用 2 字节，所消耗的空间瞬间增长了一倍。在存储空间还很宝贵的时代，这无疑是不小的负担。所以，UTF-8 编码才能够大行其道。

与 UTF-16 类似，UTF-8 编码也是一种变长传输编码。与 UTF-16 不同的是，它的编码单位不是 2 字节，而是 1 字节，也就是说，UTF-8 编码的字符，最少只用 1 字节。这样不但兼容原本就是单字节的 ASCII 编码，存储英文资料时也节省了大量的空间。

UTF-8 的转换规则及举例见表 7-2 和表 7-3。

表 7–2　UTF–8 转换规则

码值	UTF–8 编码值
U+00000000 – U+0000007F	0xxxxxxx
U+00000080 – U+000007FF	110xxxxx 10xxxxxx
U+00000800 – U+0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U+00010000 – U+001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

表 7–3　UTF–8 转换举例

码值	二进制码值	UTF–8 二进制编码值	UTF–8 编码值
U+0024	0000 0000 0010 0100	0010 0100	24
U+20AC	0010 0000 1010 1100	11100010 10000010 10101100	C2A2
U+10437	0001 0000 0100 0011 0111	1101 1000 0000 0001 1101 1100 0011 0111	D801 DC37

因为大部分中文字符都落在 0800 到 FFFF 这个区间，所以在 UTF-8 编码格式下，一个中文字符要占用 3 字节。而在 UTF-16 编码格式下，一个中文字符只占用 2 字节，看起来节省了 1 字节，但 UTF-8 凭借在传输英文资料时节省大量空间、直接与 ASCII 兼容等优势，赢得了越来越多的开发者。如今，除了少数系统内部使用 UTF-16，大部分场合，UTF-8 都已经成了默认传输标准。

需要再次强调，无论 UTF-8 还是 UTF-16，都只是传输编码。如果我们需要在代码中处理

Unicode 字符，一定是指定它原始的码值，而不是传输编码。比如“正”字，在各种编程语言中，我们都是通过 U+6B63 这个码值来指定它，而不需要考虑它在 UTF-8 或者 UTF-16 下的实际编码值。可以做一个不那么严谨的类比：码值就像身份证号，UTF-8、UTF-16、UCS-2 等传输编码就像指纹识别、声纹识别、人脸识别等方式，无论采用什么识别方式，最终定位的人，身份证号码是不变的。所以无论采用什么传输编码，无论一个中文字符要占用几字节，我们总可以用`[\u4E00-\u9FFF]`匹配中文字符。

刚制订 Unicode 规范的时候，大家乐观地认为，BMP 的空间已经足够，所以诞生了许多乐观的约定。比如在许多语言中都可以用`\u`*xxxx* 的方式来指定 Unicode 字符，其中 *xxxx* 是 4 位十六进制数。当时这样规定确实没有问题，但随着 Unicode 分配的字符不断增加，4 位十六进制数的表示法就不够用了。如果直接增加位数，可能会产生二义性问题，比如`\u10437`，到底是表示码值为 10 437 的字符呢，还是码值为 1043 的字符再加 7 这个字符呢？

所以，JavaScript 的 ECMAScript 2015 引入了新的转义序列，用`\u{`*xxxx*`}`取代了`\u`*xxxx*，这样`\u20AC` 可以写作`\u{20AC}`，`\u10437` 则写作`\u{10437}`，不会产生歧义。PHP 也是如此，在 PHP 7 里引入了`\u{`*xxxx*`}`的转义序列。而 Java 程序员就没有那么好的运气了，不能直接在 String 里面用`\u`*xxxx* 的转义序列，只能调用 `Character.toChars(10437)`来指定字符。C#程序员稍微幸运一些，以前的`\x`*n[n][n][n]*和`\u`*xxxx* 转义序列都只能支持 4 位，但`\U` 转义序列支持 8 位，所以可以写作`\U00010437`，虽然长了一些，毕竟不算太麻烦。在 Objective-C 中也是这样处理。

7.2 关于编码

通常，英文编码较为统一，都采用ASCII编码[1]或可以兼容ASCII编码（即编码表的前 127 位与ASCII编码一致，常见的各种编码，包括Unicode编码都是如此）。也就是说，英文字母、阿拉伯数字、英文的各种符号，在不同编码下的**码值**基本是一样的[2]，比如字母A，其码值总是 41；中文的情况则不同，常见的中文编码有GB18030（也就是CP54936，主要是在Windows平台下使用。早期是GBK[3]，也就是CP936，如今采用的GB18030 与GBK是兼容的，考虑到大家习惯说“GBK编码”，下文也沿用“GBK编码”的说法）和Unicode（主要用于Linux/UNIX、Mac OS）两种，同一个中文字符在两种编码下的码值并不相同。比如“正”，在GBK编码下的码值为D5FD，而在Unicode下的码值为 6B63。

为方便下面的讲解，这里先约定两种提法：

---

[1] 严格说起来应该是“Unicode 字符集”而不是“Unicode 编码”。关于编码的详细说明，也可以参考《松本行弘的程序世界》一书的第 7 章（人民邮电出版社，2011 年 8 月）。

[2] 也有例外，比如 EBCDIC 编码就是与 ASCII 不兼容的单字节编码，但我们遇到的机会很少。

[3] 如今真正的国标是 GB2312 编码，GBK 编码是对它的扩充，Windows 系列操作系统采用 GBK 编码。

ASCII 字符，即 ASCII 编码表中的字符（也就是码值在 0~127 之间的字符，不包括扩展 ASCII 字符），每个字符用 1 字节表示。常见的英文字符和半角标点符号，都属于 ASCII 字符。

非ASCII字符，即ASCII编码表之外的字符，在本书中指多字节字符。中文字符属于“非ASCII字符”，它们在GBK编码中一般占用 2 字节 [1]，在UTF-8 编码中占用 3 字节。当然“非ASCII字符”包含的字符很多，不只有中文。

7.3　尽量使用 Unicode 编码

常见的正则表达式的文档都是关于英文（ASCII 字符）的，英文开发者通常也只需要处理 ASCII 字符，不需要处理中文这类多字节的字符。不过，依照处理 ASCII 字符的方式处理中文字符，就有可能出错。

GBK（实际是GB18030）是Windows环境下默认的中文编码环境 [2]，也可以笼统地说，Windows下的默认编码就是GBK，在大多数场合使用也一切正常。如果我们希望匹配“正”（GBK码值D5FD）或者“则”（GBK码值是D4F2），很自然会想到使用字符组`[正则]`。这个思路没有问题，但结果是否和我们想象的一样呢？假设字符串是“遭遇”，这个正则表达式应当是不能匹配的。那么，来看看例 7-1 的运行结果吧。

例 7–1　GBK 编码环境下的字符组会出现错误匹配

```
//竟然能够匹配
re.search(r"[正则]", "遭遇") != None       #  =>True
//匹配的结果是什么呢？
repr(re.search(r"[正则]", "遭遇").group(0))       #  '\xd4'
//这样观察，看得更清楚
re.compile(r"[正则]", re.DEBUG)
in
  literal 213
  literal 253
  literal 212
  literal 242
```

为什么能够匹配，结果是单个字节？原因如下：虽然采用了 GBK 编码，每个中文字符用 2 字节表示，但是正则表达式处理时识别不出这是“2 个多字节字符构成的字符组”，而将其视为“4 个单字节字符构成的字符组”，4 字节正好是“正”和“则”对应的 D5 FD 和 D4 F2。（上面代码

[1] 严格来说，按照 GB18030 编码，可能出现 4 字节的字符，但绝大多数中文字符都占用 2 字节。

[2] “编码环境”指的是源代码和它所处理的数据文本采用的编码，“UTF-8 编码环境”的意思是，源代码和处理的数据都采用 UTF-8 编码，而“GBK 编码环境”的意思是，源代码和处理的数据都采用 GBK 编码。

中的 213、253、212、242 正是用十进制标识的这 4 个字节）。如果不按字符来进行匹配，而是按照字节来进行，恰好都有 D4 这个字节，所以匹配结果就是 D4 这个字节。图 7-2 说明了其中的道理。

图 7–2 GBK 编码匹配出错原理

如果在 Unicode 环境下运行，是否解决了这个问题呢？再看看例 7-2 的运行结果吧。

例 7–2 Unicode 编码环境下的字符组会出现错误匹配

```
//还是能够匹配到
re.search(r"[正则]", "遭遇") != None      #  =>True
//匹配的结果是什么呢？
repr(re.search(r"[正则]", "遭遇").group(0))      #  '\xad'
//同样来仔细看看
re.compile(r"[正则]", re.DEBUG)
in
  literal 213
  literal 253
  literal 212
  literal 242
```

问题出在哪里呢？原来，在 Python 2 下，只要没有显式指定字符串为 Unicode 字符串，即便代码文件使用的是 Unicode 格式（比如 UTF-8），仍然会按照单个字节的方式来处理，如图 7-3 所示。

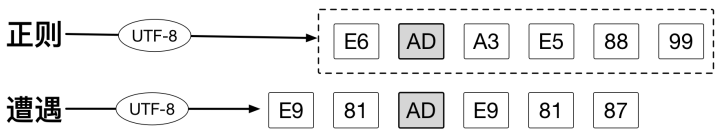


图 7–3 UTF–8 编码匹配出错原理

如果我们把正则表达式和字符串都显式指定为Unicode模式，就可以彻底解决问题了。在Python中，显式指定编码的做法是在字符串开头的引号之前写上 u 字符。请注意，如果同时用 r 指定了原生字符串，那么一定要写作 ur 而不是 ru，否则会报错。示例参见例 7-3，图示参见图 7-4。

例 7–3 彻底消灭错误匹配

```
re.search(ur"[正则]", u"遭遇") != None      #  =>True
```

```
//观察内部结构
re.compile(ur"[正则]", re.DEBUG)
literal 27491
literal 21017
```

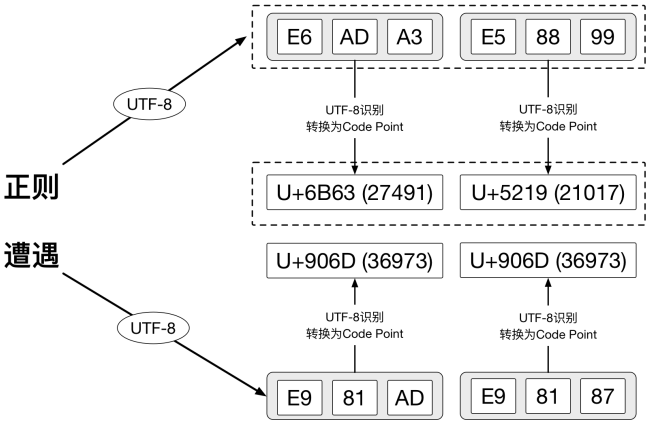


图 7–4　UTF–8 编码正确匹配

如果因为某些限制只能使用 GBK 编码，有一个“凑合能用”的办法能准确保证`[正则]`的匹配，就是把字符组`[正则]`改成多选结构`(正|则)`。从例 7-4 可以看到，此时如果要匹配成功，只能是两个连续的字节 `D5 FD` 或者 `D4 F2`，这样就避免了错误匹配。

例 7–4　GBK 编码的字符组应改为多选结构避免错误匹配

```
re.search(r"(正|则)", "遭遇") != None          #  => False
re.search(r"(正|则)", "正经") != None          #  => True
re.search(r"(正|则)", "原则") != None          #  => True
```

使用 Unicode 编码的好处还有很多，比如避免点号的错误匹配。许多文档说，点号`.`可以匹配“除换行符`\n`之外的任意**字符**”，但这可能只适用于**单字节字符**，因为点号匹配的其实只是“除换行符`\n`之外的任意**字节**”而已。

```
Python 2（GBK 编码环境）
re.search(r"^.$", "正") != None  #  =>False
PHP 4.x/5.x
preg_match("/^.$/", "正"); //  =>0，表示成功匹配的次数为 0
Ruby 1.8

"正" =~ /^.$/ != nil #  => False
```

之所以会出现这种情况，是因为正则表达式没有理解这几个字节表示的是字符“正”，仍然把它们当作互相独立的字节；既然点号`.`只能匹配 1 个字符，结果当然不能正确匹配。要解决这

个问题，必须显式指定编码，让正则表达式处理程序正确识别多字节字符。总的来说，如果使用的是Python 2、Ruby 1.8 或PHP，都需要显式指定Unicode。[1] 而在Python 3、Java、.NET和Ruby 1.9中，字符串默认采用Unicode编码，所以不存在上面的问题。

```
Python 2
re.search(r"^.$", u"正") != None      #  => True
PHP 4.x/5.x
preg_match("/^.$/u", "正")!=0;        //  => true
Ruby 1.8
"正" =~ /^.$/u                        #  => 0
```

PHP 和 Ruby 1.8 中并不存在“Unicode 字符串”，所以无法修改字符串的属性来指定 Unicode 编码。不过，如果将正则表达式指定为 Unicode 模式，正则表达式处理程序就可以正确识别字符串中的 Unicode 字符。所以如果用 PHP 或 Ruby 的正则表达式处理 Unicode 字符串，一定不要忘记指定 Unicode 模式。表 7-4 总结了常用语言中点号对多字节字符的匹配情况。

表 7–4 点号对 Unicode 字符的匹配（采用 UTF–8 编码）

字符串	正则表达式	语言	是否显式指定 Unicode 模式	可否匹配
发	`^.$`	.NET	否	可
	`^.$`	Java	否（无须指定）	可
	`^.$`	Objective-C	否（无须指定）	可
	`^.$`	JavaScript	否（无法指定）	结果取决于浏览器
	`/^.$/u`	JavaScript	是（ES2015）	可
	`/^.$/`	PHP	否	否
	`/^.$/u`	PHP	是	可
	`^.$`	Golang	否	可
	`^.$`	Python 2	否	否
	`^.$`	Python 3	否	可
	`/^.$/`	Ruby 1.8	否	否
	`/^.$/u`	Ruby 1.8	是	可
	`/^.$/`	Ruby 1.9	否	可

注 1：PHP 和 Ruby 的正则表达式本身不包含分隔符（分隔符有很多种，常见的是反斜线/），但 PHP 指定 Unicode 模式必须在后一个分隔符之后写 u，所以在这里将分隔符也写出来。

[1] 如果在 Python 2 中使用 Unicode 字符串，必须在源代码中显式说明本源文件所使用的编码，具体做法是在源代码的顶端写上这样一行：# -*- coding: utf-8 -*-。如果在 Ruby 1.8 中，则必须在源代码的顶端写上：# encoding:utf-8。

注 2：Python 2 中虽然提供了 Unicode 模式，但即便采用这种模式，点号.仍然不能匹配多字节字符，除非显式指定字符串为 Unicode 字符串（在 Python 2 中，默认是 ASCII 字符串）。

注 3：Ruby 1.9 在不同平台下的表现不一样，经过作者测试，在 Linux 下采用 Unicode 匹配规则，而在 Mac 和 Windows 中的情况通常是采用 ASCII 匹配规则。

现在来解释，虽然 GBK 编码也是多字节编码，为什么不推荐使用：GBK 编码作为常见的中文编码，使用确实很多（尤其是在 Windows 平台上），但在 GBK 编码环境下使用正则表达式，却可能遇到非常奇怪的现象，因为正则表达式处理程序一般**只能准确识别 Unicode 字符**。

总的来说，为避免出现错误匹配，只有一条原则：如果使用正则表达式处理含有多字节字符的文本，能使用 Unicode 编码环境就尽量使用。

不过，即便你使用了支持 Unicode 编码的字符串，还可能遇到各种诡异的问题。常见的例子是量词的作用问题。比如正则表达式 `ab{4}`，我们非常清楚，量词`{4}`作用的对象是字符 `b`。如果把正则表达式改为`正{4}`，问题就出现了。这时候量词作用的对象，到底是 3 个字节组合而成的“正”呢，还是“正”的最后一个字节呢？很长的时间里 JavaScript 就大受这种问题困扰，所以在 ES2015（也有人称为 ES6）中专门为正则表达式提供了 Unicode 模式，解决了这个问题。

另一个常见的问题是字符串内部的偏移值的计算单位。假如你面对的字符串是“零壹贰 34 伍陆柒”，正则表达式是`[0-9]+`，匹配的结果当然是子字符串“34”，但是它在原来字符串中的起始偏移值是多少？单从字符来看，应当是 3，因为前面有三个字符“零壹贰”。然而正确的答案是，它取决于具体的编程语言。在 Java 和 C#这样字符串原本就支持 Unicode 的语言中，起始偏移值确实是 3；在 Python 2 等语言中，如果字符串没有被指定为 Unicode 字符串，则起始偏移值是 9，因为前面三个都是中文字符，每个中文字符在 UTF-8 编码格式中需要用 3 个字符。或许会让你费解的是 Golang，虽然它的字符串也是原本就支持 Unicode 的，但偏移值却是按照字节来算的，所以起始偏移值也是 9。虽然大多数时候我们并不会关心偏移值的具体数字，只会用代码算出偏移值，再用代码来根据偏移值进行处理，但这个细节还是有必要知道的。

7.4　Unicode 与字符组简记法

上一节提到，Python 文档说明，除了指定字符串为 Unicode 字符串，还可以指定正则表达式使用 Unicode 模式，这又是怎么回事呢？

不妨回头仔细想想读过的文档，正则表达式中的`\d`和`\w`都是如何解释的？许多人的第一反应是：`\d`等价于`[0-9]`，`\w`等价于`[0-9a-zA-Z_]`，因为有些文档说明了这种等价关系。不过也有些文档说：`\d`匹配**数字字符**，`\w`匹配**单词字符**，`\s`匹配**空白字符**；但是没有说明数字字符、单词字符、空白字符到底是哪些字符。

一般来说，数字字符就是`[0-9]`，单词字符就是`[0-9a-zA-Z_]`，空白字符则包括空格、回

车等字符，但这只是 ASCII 编码中的情况，在 Unicode 编码中并非如此。

因为涵盖了多种语言和字符，所以在 Unicode 编码中，全角数字 0 、 1 、 2 之类也算作“**数字字符**”，可以由\d匹配；中文字符，也可以算作“**单词字符**”，由\w匹配；同样的道理，中文的全角空格（码值为 30FF），也可以算作“**空白字符**”，由\s匹配。所以，如果在 Python 2 中指定了正则表达式使用 Unicode 模式（最简单的方式就是在正则表达式的开头指定模式修饰符(?u)），\d、\w、\s就能匹配全角数字、中文字符、全角空格。对于这种情况，本书中称为 **Unicode 匹配规则**；相应的，之前 ASCII 编码中的匹配，称为 **ASCII 匹配规则**。例 7-5 所示的是排除型字符组的错误匹配。

例 7–5 排除型字符组的错误匹配

```
#字符均为全角
re.search(r"^\d$", u"１") != None    #  => False
re.search(r"^\w$", u"正") != None    #  => False
re.search(r"^\s$", u"　") != None    #  => False

re.search(r"(?u)^\d$", u"１") != None    #  => True
re.search(r"(?u)^\w$", u"正") != None    #  => True
re.search(r"(?u)^\s$", u"　") != None    #  => True
```

表 7-5 详细介绍了两种匹配规则下\d、\w、\s的匹配。相应的，此时\D匹配\d不能匹配的字符，\W匹配\w不能匹配的字符，\s匹配\S不能匹配的字符。

表 7–5 Unicode 匹配规则下\d、\w、\s 的匹配

简记法	ASCII 匹配规则	Unicode 匹配规则
\w	[0-9a-zA-Z_]	[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Nd}\p{Pc}]
\d	[0-9]	[\p{Nd}]
\s	\t, \n, \v, \f, \r, \x20	[\f\n\r\t\v\x85\p{Z}]

注： \p{L}表示任意语言中的字母字符（包括英文字母和汉字）。
\p{M}表示用来与其他字符结合的字符（声调、元音变化符等）。
\p{Nd}表示任何书写系统中的 0~9 的字符，汉字全角字符１、２等也算。
\p{Nl}表示形如字符的数字，比如罗马数字。
\p{Pc}表示类似下画线之类的标点字符。
\p{InEnclosedAlphanumerics}表示被包围的数字或字符，比如①。
\p{So}表示数字符号、货币符号、组合字符之外的符号字符。
具体细节请参考本章末 Unicode Property 和 Unicode Block 的详细列表。

有时候，这样的规定确实让人抓狂：假设你希望用正则表达式\d{6,12}来验证一个长度在 6 到 12 之间的数字字符串，却没留意\d能匹配全角数字，验证就可能出错；所以，一定要注意

此类问题，表 7-6 总结了常用语言中的匹配规则。

表 7–6　常用语言中的匹配规则

语言	字符组简记法的匹配规则
.NET	默认采用 Unicode 匹配规则，但可以显式指定采用 ASCII 匹配规则
Java	ASCII 匹配规则
JavaScript	`\d` 和 `\w` 采用 ASCII 匹配规则，`\s` 采用 Unicode 匹配规则
PHP	ASCII 匹配规则
Python 2	默认采用 ASCII 匹配规则，但可以显式指定采用 Unicode 匹配规则
Python 3	默认采用 Unicode 匹配规则，但可以显式指定采用 ASCII 匹配规则
Ruby 1.8	默认采用 ASCII 匹配规则，显式指定 Unicode 模式之后，只有 `\w` 使用 Unicode 匹配规则
Ruby 1.9	Unicode 匹配规则
Objective-C	默认采用 Unicode 匹配规则
Golang	ASCII 匹配规则

注 1：一般来说，单词边界\b 能匹配的位置是：一端是单词字符，一端不是单词字符（也可以什么都没有），其中单词字符的规定与\w 一样。Java 中则不是这样，它的细节比较复杂，具体请参考 Java 的章节。

注 2：在 Python 3 中可以在表达式最开始用(?a)指定 ASCII 模式。

注 3：本书中说的 Ruby 1.9 指的是 Ruby 1.9.1 及以上版本，并不包含 Ruby 1.9.0，因为这个版本使用非常少，而且有一些非常奇怪的表现。

7.5　规范化问题

Unicode 规范中专门辟出篇幅来谈“规范化等价关系”（Canonical Equivalence），这是一个重要的概念，也会影响正则表达式的匹配，所以值得单独谈一谈。

所谓“规范化”，按照维基百科的定义，指的是这样一个过程：将有多种表现形式的数据转换到“标准”的表现形式。[1]

如果这个解释不好理解，不妨来看一个例子：K。它既可以是 ASCII 字符集中的字母 K（码值 `U+0075`），也可以是绝对温度单位开尔文（Kelvin）的标志（码值 `U+212A`）。单纯从码值上来看，两者完全不相同，但如果在文本里按形状来辨识，是很难准确区分的。无论我们阅读还是写作，如果要用到开尔文这个单位，我们的第一反应还是 K。字符仍然是这个形状，只是在不同的语境中，它被赋予了不同的含义。“规范化”为开尔文标志和字母 K 定义了统一的表现形式，两

[1] Canonical Equivalence 是一个复杂的问题，具体的规定细致而完整。限于篇幅，本章不做太深入的讲解。有兴趣的读者可以参考 https://en.wikipedia.org/wiki/Unicode_equivalence。

者是“规范化等价关系”。

有了规范化等价关系，处理 Unicode 文本时就可以直接按照“看起来像”的样子来编写和使用正则表达式，而不必费心去猜测某个字符到底是什么码值。

在正则表达式中，一般默认不会开启对规范化等价关系的支持，需要显式指定，见例 7-6。比如在 JavaScript 中，在同时指定了 Unicode 匹配规则和不区分大小写匹配模式的情况下，会开启对规范化等价关系的支持，见例 7-6。

例 7–6　JavaScript 中的规范化等价关系

```
#大小写字母均可以判断等价
/K/iu.test('\u{212A}');  //=> true
/k/iu.test('\u{212A}');  //=> true
```

而在 Java 中，是通过指定常量 `Pattern.CANON_EQ` 来开启的（虽然按照 JDK 的文档，这么做会严重降低性能），见例 7-7。

例 7–7　Java 中的规范化等价关系

```
Pattern upperK = Pattern.compile("K", Pattern.CANON_EQ);
upperK.matcher("Unicode Kelvin sign \u212a").find();      //=> True

Pattern lowerK = Pattern.compile("k", Pattern.CANON_EQ);
lowerK.matcher("Unicode Kelvin sign \u212a").find();      //=> True
```

7.6 单词边界

单词边界之前已经介绍过（☞ 60），它的准确解释是：一端必须出现`\w`能匹配的字符，另一端不出现`\w`能匹配的字符。[1] 在JavaScript、PHP、Python 2、Ruby中，`\w`只能匹配`[0-9a-zA-Z_]`。所以在这些语言中，`\b\w+\b`能用来匹配几乎所有的英文单词，在例 7-8 中用它来提取一段文本中的所有单词。

例 7–8　用单词边界提取单词

```
re.findall(r"\b\w+\b", "a sentence contains a lot of words")
["a", "sentence", "contains", "a", "lot", "of", "words"]
```

之所以说“几乎所有”的英文单词，是因为有些情况更加复杂：假设字符串中包含中文，之间用空格字符分隔，比如“他们 我们去见他们的时候……”。如果希望匹配单独出现的“他们”，

[1] Objective-C 是例外，默认情况下，它的字符组简记法和单词边界都采用 Unicode 规则。如果显式指定 ASCII 匹配模式，受影响的只有单词边界，字符组简记法的匹配并不会发生变化。

最自然的想法似乎是使用正则表达式`\b 他们\b`，结果却是不对的，如例 7-9 所示。

例 7–9　用单词边界提取中文单词会出错

```
re.findall(r"\b 他们\b", "他们 我们去见他们的时候……")
[]
```

原因在于，默认情况下，`\w`是按照 ASCII 匹配规则的。“他”字之前的位置，左侧没有字符，不能由`\w`匹配，右侧的“他”字也不能由`\w`匹配；“们”字右侧的情况也是如此。所以，`\b 他们\b`无法匹配“他们”。要解决这个问题可以采用 Unicode 匹配规则，此时`\w`就不只能匹配`[0-9a-zA-Z_]`，还能匹配其他语言中的单词字符（包括中文字符），程序代码如例 7-10 所示。

例 7–10　在 Unicode 模式下提取中文单词

```
re.findall(r"(?u)\b 他们\b", "他们 我们去见他们的时候……")
['他们']
```

但是，这样又出现了新的问题。比如字符串“学习 regex”，其中的 `regex` 在非 Unicode 匹配规则下是可以由`\bregex\b`匹配的；在 Unicode 匹配规则下却行不通，因为此时`\w`既能匹配中文字符又能匹配英文字符，`\bregex\b`反而无法匹配 `regex` 了，代码如例 7-11 所示。

例 7–11　在 Unicode 模式下提取单词仍然有问题

```
re.findall(r"\bregex\b", "学习 regex")
["regex"]
re.findall(r"(?u)\bregex\b", "学习 regex")
[]
```

为了形象说明在 ASCII 匹配规则和 Unicode 匹配规则下`\b`能够匹配的“单词边界”（因为通常处理的文本包含中文和英文，所以只是简要列出了各分类的常用字符），下面用图 7-5 和图 7-6 分别说明。其中深色区域是必须出现的，而浅色的区域是可以出现也可以不出现的（比如文本首尾位置的空字符串）。

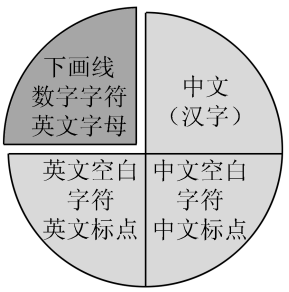


图 7–5　ASCII 匹配规则下的单词边界

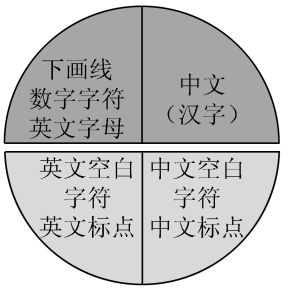


图 7–6　Unicode 匹配规则下的单词边界

表 7-7 举列列出了在 Unicode 匹配模式下，\b匹配的几种典型的场景。

表 7–7 在 Unicode 匹配模式下，单词边界的几种典型场景

字符串	正则表达式	能否匹配	说明
tom，jerry（半角标点）	\btom\b	能	\w不能匹配半角标点
tom，jerry（全角标点）	\btom\b	否	\w可以匹配全角标点
tom 和 jeryy	\b 和\b	否	\w可以匹配中文字符
汤姆,jerry（半角标点）	\b 汤姆\b	能	\w不能匹配半角标点
汤姆，jerry（全角标点）	\b 汤姆\b	否	\w可以匹配全角标点

总的来说，如果使用 Unicode 匹配规则，**尽量不要**在处理中英文混排文本时使用\b。如果使用 ASCII 匹配规则，则可以在处理英文文本时放心地使用\b。

也有更复杂的情况，比如 Java 就是如此。在 Java 中，虽然\w只能匹配[0-9a-zA-Z_]，\b对“单词字符”的判断却是按照 Unicode 匹配规则的。\b的匹配问题，本书后面关于各种语言的章节中都会专门讲解。

7.7 码值转义序列

Unicode 字符多种多样，除去 ASCII 中的字母、数字、标点和中文字符，还包括其他多种语言和多种符号，有些符号甚至很难打出来（比如表示商标注册的™），这时候该如何表示呢？再说远一点，如果我们想用一个字符组匹配所有的中文字符，能不能像[a-z]那样呢？

使用正则表达式解决这类问题，必须依赖码值。前面讲过，每一个 Unicode 字符都有一个 Unicode 码值，所以在正则表达式中的 Unicode 字符往往采用 Unicode 码值来指定。

一般来说，指定码值的形式有两种：\u*xxxx* 和\u{*xxxx*}（其中的 *xxxx* 为编码的值，\u 之后必须有 4 位十六进制数字）。.NET、Java、JavaScript、Objective-C 和 Python 使用前一种形式，而 PHP 和 Ruby 使用后一种形式（Ruby 1.9 以上版本才支持这种表示法，☞280；PHP 使用的字母是 x 而不是 u：\x{*xxxx*}）。

比如“正则”的“正”字对应的 Unicode 编码是 U+6B63，所以可以在.NET、Java、JavaScript 的正则表达式中用\u6B63 表示它；Python 稍有不同，必须使用 u"\u6B63"（之前的 u 表示这是一个 Unicode 字符串，☞213）；在 Ruby 中必须写作\u{6B63}；在 PHP 中则写作\x{6B63}。表 7-8 总结了常用语言中用码值表示 Unicode 字符的方法。通过前面的介绍我们知道，我们用到的 Unicode 字符不限于 BMP，所以码值可能大于 U+FFFF，也就是说，4 位十六进制数是不够的。如果采用\u*xxxx* 表示法，通常就需要另想办法来表示此字符，否则会有二义性问题，如果采用\u{*xxxx*}表示法就没有这种问题。

表 7–8　常用语言中用码值表示 Unicode 字符的方法

编码	语言	表示法	说明
6B 63	.NET	`\u6B63`	
	Java	`\u6B63`	
	JavaScript	`\u6B63`	
	JavaScript	`\u{6B63}`	ECMAScript 2015 支持
6B 63	Python	`\u6B63`	必须使用 Unicode 字符串，在 Python 2 中，要在字符串之前加 u 指定 Unicode 字符串
	PHP	`\x{6B63}`	必须指定 Unicode 模式
	Ruby	`\u{6B63}`	限 Ruby 1.9 以上版本，且必须显式指定 Unicode 模式
	Objective-C	`\u6B63`	
	Golang	`\u6B63`	

既然可以这样指定Unicode字符，自然也可以在字符组中用范围表示法（☞6）指定Unicode编码范围。这个功能最常见的应用就是匹配任意一个中文字符。查询Unicode编码表可知，中文字符的码值大多位于 `4E00` 到 `9FFF`之间 [1]，所以可以用字符组匹配中文字符（Unicode编码`4E00`~`9FFF`归类为**CJK 统一表意符号CJK Unified Ideographs**[2]，涵盖了绝大多数中文字符）。表 7-9 列出了常用语言中匹配中文字符的字符组。

表 7–9　常用语言中匹配中文字符的字符组

语言	字符组	说明
.NET	`[\u4E00-\u9FFF]`	
Java	`[\u4E00-\u9FFF]`	
JavaScript	`[\u4E00-\u9FFF]`	
	`[\u{4E00}-\u{9FFF}]`	这种写法无疑更清楚，但必须显式指定正则表达式使用 Unicode 模式
Python	`[\u4E00-\u9FFF]`	必须使用 Unicode 字符串，在 Python 2 中，要在字符串之前加 u 指定 Unicode 字符串

[1] 网上有许多资料称中文字符所处的字符范围在 U+4E00~U+9FA5 之间，而不是 U+4E00~U+9FFF，这种说法有一定的道理，因为在 1992 年提交给 IRG（International Rapporteur Group）的字符只排到 U+9FA5。然而在这之后，制订 Unicode 4.2、5.1 和 5.2 规范时都进行了扩展，新增了字符。但是，U+4E00~ U+9FFF 是预留给东亚文字的编码点，所以使用 4E00~9FFF 是更好的选择。具体信息可以参考 http://www.unicode.org/versions/Unicode6.0.0/ch12.pdf。

[2] 请参考 http://en.wikipedia.org/wiki/CJK_Unified_Ideographs。

（续表）

语言	字符组	说明
PHP	`[\x{4E00}-\x{9FFF}]`	必须指定 Unicode 模式
	`[\u{4E00}-\u{9FFF}]`	PHP 7 以上版本中可用
Ruby	`[\u{4E00}-\u{9FFF}]`	限 Ruby 1.9 以上版本
Objective-C	`[\u4E00-\u9FFF]`	
Golang	`[\u4E00-\u9FFF]`	

如果实在没有办法使用 Unicode 编码环境，而只能采用 GBK 编码，在 Python、Ruby 1.8 和 PHP 中，也有一个办法匹配所有中文字符：查阅 GBK 编码表可知，中文字符的 GBK 码值从 `B0 00` 开始，到 `FE A0` 结束，但是在通常情况下正则引擎无法正确识别非 Unicode 编码字符串的字符边界，所以不能写作`[\xb000-\xfea0]`，只能写作`[\xb0-\xfe][\x00-\xff]`，它匹配两个字节，第一个字节的码值范围从 `B0` 到 `FE`，第二个字节的码值范围从 `00` 到 `FF`，如果要匹配多个中文字符，必须添加括号将这两个字节分为一组，再使用量词，比如`([\xb0-\xfe][\x00-\xff])+`。因为此时完全是将字符串作为单字节序列来对待的，所以也不应该指定任何与 Unicode 有关的设置（无论是正则表达式还是字符串，都是如此）。

7.8 Unicode 属性

每一个 Unicode 字符，除了有 Code Point 与之对应外，还具有其他属性，在正则表达式中常用到 3 种 Unicode 属性：Unicode Property、Unicode Block、Unicode Script，分别对应字符的功能、所属代码区段、书写系统；它们的表现形式类似`\p{property}`。图 7-7 比较形象地说明了这 3 种属性，下面详细进行介绍（本节介绍的 Unicode 字符均限于 BMP）。

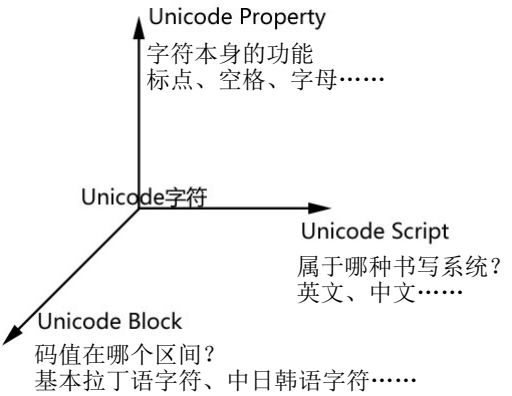


图 7–7 Unicode 字符的三种属性

7.8.1　Unicode Property

Unicode Property 的记法类似`\p{L}`、`\p{P}`。它按照字符的功能分类 Unicode 字符，每个 Unicode 字符只能属于一个 Unicode Property。

可以这样理解 Unicode Property：它并不关心字符所属的语言，只关心字符的功能，比如`\p{Z}`表示任意的空白字符或不可见的分隔符；`\p{P}`表示任何标点字符，等等。遇到中英文混排、全角、半角字符同时出现的情况，就可以用`\p{Z}`匹配所有的空白字符（而不用关心空格到底是全角空格还是半角空格），用`\p{P}`匹配所有的标点字符（而不用关心逗号到底是中文逗号还是英文逗号）。可以说，有了 Unicode Property，各种字符“第一次”可以按功能（或者意义）来分类，不再是一堆零散的码值。正因为如此，大多数语言中的正则表达式，如果不是支持所有的 Unicode 属性，最起码也会支持 Unicode Property。

如果把 Unicode Property 理解为一个“字符组”，那么一定还有对应的排除型字符组，此排除型字符组的通行记法是将`\p{property}`中的小写 p 改为大写 P，写作`\P{property}`。这样，`\P{Z}`对应`\p{Z}`无法匹配的字符，`\P{P}`对应`\p{P}`无法匹配的字符。Unicode Block 和 Unicode Script 对应的排除型字符组也是这样标记，下面不再赘述。

支持 Unicode Property 的语言有.NET、Java、Objective-C、JavaScript（ES2017）、PHP、Ruby（限 1.9 以上版本）、Golang，在 PHP 和 Ruby 中使用 Unicode Property 时，必须要开启 Unicode 模式。下面的代码展示了在几种常用语言中，`\p{P}`如何匹配一个全角逗号。

```
.NET
Regex.IsMatch("，", "\\p{P}");   //  =>True

Java
"，".matches("\\p{P}");          //  => True

PHP
preg_match("/\\p{P}/u", "，");   // => 1

Ruby 1.9
/\p{P}/u =~ "，"    #  => 0
```

Unicode Property 的列表附在章末。

7.8.2　Unicode Block

Unicode Block 不同于 Unicode Property，它按照编码区间划分 Unicode 字符，各个区间彼此联系但互不相交，所以每个 Unicode 编码中的每个字符都有唯一归属的 Unicode Block。

在 Unicode 编码表中，同一种语言的字符通常是落在同一区间的，所以 Unicode Block 也可以粗略表示某类语言的字符，比如`\p{InHebrew}`表示希伯来语字符，`\p{InCJK`

Unified_Ideographs}表示兼容 CJK（中文、日文、韩文）统一表意字符。如果读者细心阅读文档，会发现 Unicode Block 的名字虽然类似某种语言的名字，但都有 In（Java 风格）或者 Is（.NET 风格）前缀，所以它对应的还是“落在某个区间的 Unicode 字符”。

在本书介绍的语言中，Java、.NET、Golang 支持 Unicode Block，但是记法不相同，下面以 CJK 统一表意字符为例，虽然记法不同，但功能是相同的，都可以匹配任意一个中文字符。

```
Java
"我".matches("\\p{InCJK_Unified_Ideographs}");          // => True

.NET
Regex.IsMatch("我", @"\p{IsCJKUnifiedIdeographs}");     // => True
```

另外值得一用的是\p{InCJK_Symbols_and_Punctuation}，从命名上看，它覆盖了中文、日文、韩文中的大部分常见标点，但事实并非如此，比如全角逗号，就不在这个 Block 中。不过，所有标点都在\p{P}这个 Unicode Property 中，代码如例 7-12 所示。

例 7–12　全角逗号不属于\p{InCJK_Symbols_and_Punctuation}

```
"。".matches("\\p{InCJK_Symbols_and_Punctuation}");    // => True
"，".matches("\\p{InCJK_Symbols_and_Punctuation}");    // => False
"。".matches("\\p{P}");     // => True
"，".matches("\\p{P}");     // => True
```

Unicode Block 的列表附在章末。

7.8.3 Unicode Script

Unicode Script 按照字符所属的书写系统来划分 Unicode 字符，比如\p{Greek}表示希腊语字符，\p{Han}表示汉语（中文字符）。它的写法类似 Unicode Block，只是名字的开头没有 Is 或者 In。

在本书介绍的语言中， PHP、Ruby（限 1.9 以上版本）、Objective-C[1]支持Unicode Script，PHP在使用Unicode Script时，必须开启Unicode模式（详见PHP的章节）。在这几种语言中，我们可以很方便地用\p{Han}来匹配中文字符。

```
PHP
preg_match("/\\p{Han}/u", "我");     // => 1
Ruby 1.9
/\p{Han}/u =~ "我"       # => 0
```

Unicode Script 的列表附在章末。

[1] Objective-C 的文档中虽然没有说明，但实际测试发现可以正常支持。

7.9　Unicode 属性列表

7.9.1　Unicode Property

每个 Unicode 字符都只能属于一个 Unicode Property。BMP 中所有的 Unicode Property 共分为 7 大类，30 小类。大类的名字只有单个字母，小类的名字则包含多个字母，开头字母与所在大类的名字相同，小类包含的字符都属于它所在的大类。表 7-10 列出了所有的 Unicode Property。

表 7–10　Unicode Property

Unicode Property		说明
\p{C}		不可见的控制字符和未使用的码值
	\p{Cc}	ASCII 编码中，0x00 到 0x1F 或 Latin-1 编码中 0x80 到 0x9F 的控制字符
	\p{Cf}	不可见的格式字符
	\p{Co}	留作私用的码值
	\p{Cs}	UTF-16 编码中 surrogate pair 的一半
	\p{Cn}	未指定的码值
\p{L}		各种语言中的字母
	\p{Ll}	具有大写形式的字母的小写形式
	\p{Lt}	只有在单词首位才大写的字符
	\p{L&}	等于 Ll、Lu、Lt 的组合
	\p{Lo}	没有大小写形态的字符
	\p{Lu}	具有小写形式的字母的大写形式
\p{M}		用来与其他字符结合的字符（声调、元音变化符等）
	\p{Mc}	与其他字符组合，并且会占用空间的字符（常见于东亚语言）
	\p{Me}	需要成对出现的字符，比如圆括号、方括号
	\p{Mn}	用来与其他字符结合，但并不占用额外空间的字符
\p{N}		各种书写系统中的数字字符
	\p{Nd}	各种书写系统中的 0~9 字符
	\p{Nl}	形如字符的数字，比如罗马数字
	\p{No}	上标或者下标数字，或者是 0~9 之外的数字（不包括表意书写系统中的数字）

（续表）

Unicode Property		说明
\p{P}		各种标点符号
	\p{Pd}	各种连字符号
	\p{Ps}	成对但不同的符号的前半部分（包括英文括号、中文括号、书名号）
	\p{Pe}	成对但不同的符号的后半部分（包括英文括号、中文括号、书名号）
	\p{Pi}	成对且相同的符号的前半部分（比如单引号、双引号）
	\p{Pf}	成对且相同的符号的后半部分（比如单引号、双引号）
	\p{Pc}	类似下画线之类的标点字符
	\p{Po}	除横线、括号、引号和连接符之外的任何标点符号
\p{S}		数字符号、货币符号
	\p{Sm}	数字符号
	\p{Sc}	货币符号
	\p{Sk}	由多个字符构成的组合字符
	\p{So}	数字符号、货币符号和组合字符之外的符号字符
\p{Z}		空白字符，或者不可见的分隔符
	\p{Zs}	不可见但占用空间的空白字符
	\p{Zl}	分行符 U+2028
	\p{Zp}	分段符 U+2029

7.9.2 Unicode Block

每个 Unicode Block 都对应一个连续的 Unicode 码值区间，BMP 中的字符一共划分为 105 个 Block。

使用时应该注意，Java使用的Unicode Block是\p{In…}形式的 [1]，比如InCJK_Unified_Ideographs；而.NET使用的Unicode Block是\p{Is…}形式的，同时不会包含下画线 [2]，比如IsCJKUnifiedIdeographs。表 7-11 列出了所有的Unicode Block，但没有包含Is或者In的前缀，使用时请自行添加。

[1] 具体情况可参考 http://download.oracle.com/javase/6/docs/api/java/lang/Character.UnicodeBlock.html。

[2] 具体情况可参考 http://msdn.microsoft.com/en-us/library/20bw873z.aspx#SupportedNamedBlocks。

表 7–11　Unicode Block

码值	Unicode Block	说明
U+0000~U+007F	\p{Basic_Latin}	控制符及基本拉丁文
U+0080~U+00FF	\p{Latin-1_Supplement}	控制符及拉丁文补充
U+0100~U+017F	\p{Latin_Extended-A}	拉丁文扩展-A
U+0180~U+024F	\p{Latin_Extended-B}	拉丁文扩展-B
U+0250~U+02AF	\p{IPA_Extensions}	国际音标扩展
U+02B0~U+02FF	\p{Spacing_Modifier_Letters}	空白修饰字符
U+0300~U+036F	\p{Combining_Diacritical_Marks}	结合用读音符号
U+0370~U+03FF	\p{Greek_and_Coptic}	希腊文及科普特文
U+0400~U+04FF	\p{Cyrillic}	西里尔字母
U+0500~U+052F	\p{Cyrillic_Supplementary}	西里尔字母补充
U+0530~U+058F	\p{Armenian}	亚美尼亚语
U+0590~U+05FF	\p{Hebrew}	希伯来语
U+0600~U+06FF	\p{Arabic}	阿拉伯语
U+0700~U+074F	\p{Syriac}	叙利亚语
U+0780~U+07BF	\p{Thaana}	马尔代夫语
U+0900~U+097F	\p{Devanagari}	天城文书
U+0980~U+09FF	\p{Bengali}	孟加拉语
U+0A00~U+0A7F	\p{Gurmukhi}	锡克教文
U+0A80~U+0AFF	\p{Gujarati}	古吉拉特文
U+0B00~U+0B7F	\p{Oriya}	奥里亚文
U+0B80~U+0BFF	\p{Tamil}	泰米尔文
U+0C00~U+0C7F	\p{Telugu}	泰卢固文
U+0C80~U+0CFF	\p{Kannada}	卡纳达文
U+0D00~U+0D7F	\p{Malayalam}	德拉维族语
U+0D80~U+0DFF	\p{Sinhala}	僧伽罗语
U+0E00~U+0E7F	\p{Thai}	泰语
U+0E80~U+0EFF	\p{Lao}	老挝语
U+0F00~U+0FFF	\p{Tibetan}	藏语
U+1000~U+109F	\p{Myanmar}	缅甸语
U+10A0~U+10FF	\p{Georgian}	格鲁吉亚语
U+1100~U+11FF	\p{Hangul_Jamo}	朝鲜语
U+1200~U+137F	\p{Ethiopic}	埃塞俄比亚语

（续表）

码值	Unicode Block	说明
U+13A0~U+13FF	\p{Cherokee}	切诺基语
U+1400~U+167F	\p{Unified_Canadian_Aboriginal_Syllabics}	统一加拿大土著语音节
U+1680~U+169F	\p{Ogham}	欧甘字母
U+16A0~U+16FF	\p{Runic}	如尼文
U+1700~U+171F	\p{Tagalog}	塔加拉语
U+1720~U+173F	\p{Hanunoo}	Hanunóo
U+1740~U+175F	\p{Buhid}	Buhid
U+1760~U+177F	\p{Tagbanwa}	Tagbanwa
U+1780~U+17FF	\p{Khmer}	高棉语
U+1800~U+18AF	\p{Mongolian}	蒙古语
U+1900~U+194F	\p{Limbu}	Limbu
U+1950~U+197F	\p{Tai_Le}	德宏泰语
U+19E0~U+19FF	\p{Khmer_Symbols}	高棉语记号
U+1D00~U+1D7F	\p{Phonetic_Extensions}	语音学扩展
U+1E00~U+1EFF	\p{Latin_Extended_Additional}	拉丁文扩充附加
U+1F00~U+1FFF	\p{Greek_Extended}	希腊语扩展
U+2000~U+206F	\p{General_Punctuation}	常用标点
U+2070~U+209F	\p{Superscripts_and_Subscripts}	上标及下标
U+20A0~U+20CF	\p{Currency_Symbols}	货币符号
U+20D0~U+20FF	\p{Combining_Diacritical_Marks_for_Symbols}	组合用记号
U+2100~U+214F	\p{Letterlike_Symbols}	字母式符号
U+2150~U+218F	\p{Number_Forms}	数字形式
U+2190~U+21FF	\p{Arrows}	箭头
U+2200~U+22FF	\p{Mathematical_Operators}	数学运算符
U+2300~U+23FF	\p{Miscellaneous_Technical}	杂项工业符号
U+2400~U+243F	\p{Control_Pictures}	控制图片
U+2440~U+245F	\p{Optical_Character_Recognition}	光学识别符
U+2460~U+24FF	\p{Enclosed_Alphanumerics}	封闭式字母和数字
U+2500~U+257F	\p{Box_Drawing}	制表符号
U+2580~U+259F	\p{Block_Elements}	方块元素
U+25A0~U+25FF	\p{Geometric_Shapes}	几何图形
U+2600~U+26FF	\p{Miscellaneous_Symbols}	杂项符号

（续表）

码值	Unicode Block	说明
U+2700~U+27BF	\p{Dingbats}	印刷符号
U+27C0~U+27EF	\p{Miscellaneous_Mathematical_Symbols-A}	杂项数学符号 A
U+27F0~U+27FF	\p{Supplemental_Arrows-A}	箭头补充 A
U+2800~U+28FF	\p{Braille_Patterns}	盲点文字模型
U+2900~U+297F	\p{Supplemental_Arrows-B}	箭头补充 B
U+2980~U+29FF	\p{Miscellaneous_Mathematical_Symbols-B}	杂项数学符号 B
U+2A00~U+2AFF	\p{Supplemental_Mathematical_Operators}	数学运算符补充
U+2B00~U+2BFF	\p{Miscellaneous_Symbols_and_Arrows}	杂项箭头和符号
U+2E80~U+2EFF	\p{CJK_Radicals_Supplement}	CJK 部首补充
U+2F00~U+2FDF	\p{Kangxi_Radicals}	康熙字典部首
U+2FF0~U+2FFF	\p{Ideographic_Description_Characters}	表意文字描述符
U+3000~U+303F	\p{CJK_Symbols_and_Punctuation}	CJK 标点和符号
U+3040~U+309F	\p{Hiragana}	日文平假名
U+30A0~U+30FF	\p{Katakana}	日文片假名
U+3100~U+312F	\p{Bopomofo}	注音字母
U+3130~U+318F	\p{Hangul_Compatibility_Jamo}	朝鲜文兼容字母
U+3190~U+319F	\p{Kanbun}	象形字注释标注
U+31A0~U+31BF	\p{Bopomofo_Extended}	注音字母扩展
U+31F0~U+31FF	\p{Katakana_Phonetic_Extensions}	日文片假名语音扩展
U+3200~U+32FF	\p{Enclosed_CJK_Letters_and_Months}	封闭式 CJK 文字和月份
U+3300~U+33FF	\p{CJK_Compatibility}	CJK 兼容
U+3400~U+4DBF	\p{CJK_Unified_Ideographs_Extension_A}	CJK 统一表意符号扩展 A
U+4DC0~U+4DFF	\p{Yijing_Hexagram_Symbols}	易经六十四卦符号
U+4E00~U+9FFF	\p{CJK_Unified_Ideographs}	CJK 统一表意符号
U+A000~U+A48F	\p{Yi_Syllables}	彝文音节
U+A490~U+A4CF	\p{Yi_Radicals}	彝文字根
U+AC00~U+D7AF	\p{Hangul_Syllables}	朝鲜文音节
U+D800~U+DB7F	\p{High_Surrogates}	UTF-16 高半区域
U+DB80~U+DBFF	\p{High_Private_Use_Surrogates}	自行使用的 UTF-16 高半区域
U+DC00~U+DFFF	\p{Low_Surrogates}	UTF-16 低半区域
U+E000~U+F8FF	\p{Private_Use_Area}	自行使用区域
U+F900~U+FAFF	\p{CJK_Compatibility_Ideographs}	CJK 兼容象形文字

（续表）

码值	Unicode Block	说明
U+FB00~U+FB4F	\p{Alphabetic_Presentation_Forms}	字母表达形式
U+FB50~U+FDFF	\p{Arabic_Presentation_Forms-A}	阿拉伯表达形式 A
U+FE00~U+FE0F	\p{Variation_Selectors}	变量选择符
U+FE20~U+FE2F	\p{Combining_Half_Marks}	组合用半符号
U+FE30~U+FE4F	\p{CJK_Compatibility_Forms}	CJK 兼容形式
U+FE50~U+FE6F	\p{Small_Form_Variants}	小型变体形式
U+FE70~U+FEFF	\p{Arabic_Presentation_Forms-B}	阿拉伯表达形式 B
U+FF00~U+FFEF	\p{Halfwidth_and_Fullwidth_Forms}	半型及全型形式
U+FFF0~U+FFFF	\p{Specials}	特殊

7.9.3 Unicode Script

除去没有赋值的 Unicode 码值之外，每个 Unicode 码值都归属某个 Unicode Script，它一般对应某种语言。表 7-12 列出了所有的 Unicode Script。

表 7–12 Unicode Script

\p{Common}	\p{Georgian}	\p{Limbu}	\p{Han}
\p{Arabic}	\p{Inherited}	\p{Malayalam}	\p{Tamil}
\p{Armenian}	\p{Greek}	\p{Mongolian}	\p{Telugu}
\p{Bengali}	\p{Kannada}	\p{Myanmar}	\p{Thaana}
\p{Bopomofo}	\p{Gujarati}	\p{Ogham}	\p{Thai}
\p{Braille}	\p{Katakana}	\p{Oriya}	\p{Tibetan}
\p{Buhid}	\p{Gurmukhi}	\p{Runic}	\p{Yi}
\p{CanadianAboriginal}	\p{Khmer}	\p{Sinhala}	
\p{Cherokee}	\p{Han}	\p{Syriac}	
\p{Cyrillic}	\p{Lao}	\p{Tagalog}	
\p{Devanagari}	\p{Hangul}	\p{Tagbanwa}	
\p{Ethiopic}	\p{Latin}	\p{TaiLe}	

7.10 POSIX 字符组

在第 1 章，我们已经介绍过 POSIX 字符组，并且提到，根据 locale 的不同，这些字符组简记法能匹配的字符也不相同，表 7-13 介绍了 Unicode 编码中这些简记法能匹配的字符。

表 7–13　Unicode 编码中的 POSIX 字符组

POSIX	说明	Unicode 属性表示法
`[:alnum:]`	字母字符和数字字符	`[\p{L&}\p{Nd}]`
`[:alpha:]`	字母	`\p{L&}`
`[:ascii:]`	ASCII 字符	`\p{InBasicLatin}`
`[:blank:]`	空格字符和制表符	`[\p{Zs}\t]`
`[:cntrl:]`	控制字符	`\p{Cc}`
`[:digit:]`	数字字符	`\p{Nd}`
`[:graph:]`	空白字符之外的字符	`[^\p{Z}\p{C}]`
`[:lower:]`	小写字母字符	`\p{Ll}`
`[:print:]`	类似`[:graph:]`，但包括空白字符	`\P{C}`
`[:punct:]`	标点符号	`[\p{P}\p{S}]`
`[:space:]`	空白字符	`[\p{Z}\t\r\n\v\f]`
`[:upper:]`	大写字母字符	`\p{Lu}`
`[:word:]*`	字母字符	`[\p{L}\p{N}\p{Pc}]`
`[:xdigit:]`	十六进制字符，等于`[0-9a-zA-Z]`	`[A-Fa-f0-9]`

注：标记*的字符组简记法并不是 POSIX 规范中的，但使用很多，一般语言中都提供了，文档也会提到。

在本书介绍的 6 种语言中，Java、PHP、Ruby 支持使用 POSIX 字符组。但是只有 Ruby 1.9 的 POSIX 字符组是完全支持 Unicode 字符的，并且不需要显式指定 Unicode 模式。在 Java 和 PHP 中，POSIX 字符组只能匹配 ASCII 字符。

7.11　Emoji

Emoji（“绘文字”）已经不再局限在日本流行，而是获得世界各国人民广泛使用。Unicode 6.0 及以上版本已经收录了 Emoji 表情，并且随着 Unicode 规范的升级，Emoji 表情也在不断更新。到本章写作时为止，最新的版本是 2017 年 5 月 18 日定稿发布的 Emoji 5.0，其中收录了 2666 个 Emoji。要注意的是，Unicode 规范中定义的 Emoji 并不都是新添加的，一些老的字符也会被算作 Emoji，比如 ASCII 编码中的#，也在官方的 Emoji 列表中。如果希望获得最全的 Emoji 列表，可以访问这个地址：http://www.unicode.org/Public/emoji/5.0/emoji-data.txt。

与常见字符不同的是，关于 Emoji，Unicode 规范中只定义了 Emoji 的意义，没有规定它的具体形态，所以“同样”的 Emoji，在不同的平台（iOS、Android、微信等）上的表现会有细微的差别。

Emoji 与常见字符相区别的另一点是，许多 Emoji 都不在 BMP 内，码值超过 FFFF，所以\u*xxxx*

的表示法无能为力，`\u{xxxx}`则没有这个问题。同样，因为其码值超过 `FFFF`，所以在 UTF-8 编码格式下需要用 4 字节——既不同于 ASCII 字符的 1 字节，也不同于中文的 3 字节。

处理 Emoji 的思路和处理传统文本的思路有很大区别。第一，Emoji 是不可读的，没有读音可言；第二，Emoji 是表意的，只能从图像的角度来理解；第三，许多时候 Emoji 对程序代码而言是不可见的，只能用`\u{xxxx}`的方式来表达，仅仅阅读源代码并不知道这个字符到底是什么；第四，传统的 Unicode 系统未必能存储 Emoji，典型的例子是 MySQL，必须将编码设定为 UTF8MB4 才可以接收包含 Emoji 的文本，“传统上”用的 UTF-8 会报错。

那么，正则表达式处理 Emoji 有什么好办法吗？网络上有一些开源的工具，其原理大都是找出 Emoji 所在的码值区间，然后用正则表达式的字符组来匹配。普通用途应当没有问题，但这些工具的作者对 Emoji 的理解深浅不一，Emoji 也在不断增加，用起来还是应该谨慎。

图 7-8 所示为 Emoji 表情。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+1F60x	😀	😁	😂	😃	😄	😅	😆	😇	😈	😉	😊	😋	😌	😍	😎	😏
U+1F61x	😐	😑	😒	😓	😔	😕	😖	😗	😘	😙	😚	😛	😜	😝	😞	😟
U+1F62x	😠	😡	😢	😣	😤	😥	😦	😧	😨	😩	😪	😫	😬	😭	😮	😯
U+1F63x	😰	😱	😲	😳	😴	😵	😶	😷	😸	😹	😺	😻	😼	😽	😾	😿
U+1F64x	🙀	🙁	🙂	🙃	🙄	🙅	🙆	🙇	🙈	🙉	🙊	🙋	🙌	🙍	🙎	🙏

图 7–8　一组常见的 Emoji 表情及对应的 Code Point

正则表达式擅长的是处理文本而不是图像，虽然`[\u{1f600}-\u{1f64f}]`这样的正则表达式是能运行的，可以匹配主要的 Emoji 表情，但这种正则表达式基本不可读。而想用一个正则表达式找到全部 Emoji 的办法也不可行：Emoji 分散在 BMP 和各个补充平面中，也不能归一到某个 Unicode Block 或是 Unicode Property。虽然有 Emoticons 这个 Unicode Block，但它只包含了 80 个 Emoji，而 Emoji 的总数有 2666 个，覆盖面太低。

实际上，不只是正则表达式处理 Emoji 很麻烦，通用的文本处理对它们也束手无策。正因为如此，许多语言都提供了与 Emoji 相关的专门的工具，有效降低了处理 Emoji 的难度。

下面的代码来自 Python 的 Emoji 包，其中的`:thumbs_up_sign:`就表示“挑大拇指”的 Emoji 👍。

```
import emoji
print(emoji.emojize('Python is :thumbs_up_sign:'))
```

第 8 章　匹配原理

在固定字符串的处理上，正则表达式的速度是赶不上简单字符串处理的；如果要进行复杂多变的字符串处理，正则表达式的速度则要胜于简单的字符串处理，比如正则表达式`a(bb)+a`，它能匹配的字符串是`abba`、`abbbba`、`abbbbbba`、…如果用简单的字符串匹配算法，我们可能需要逐个列举可能的形态，也就是`abba`、`abbbba`、`abbbbbba`、…此类尝试不知道要进行多少次，效率自然大打折扣。这不免让人好奇，正则表达式为什么这样快呢？要弄清这个问题，必须详细了解正则表达式的匹配原理。[1]

8.1　有穷自动机

正则表达式能迅速进行复杂处理的秘密在于，它采用了一种特殊的理论模型：**有穷自动机**（Finite Automata，也叫**有穷状态自动机**，finite-state machine）。这种机器具备有限个状态，可以根据不同的条件在状态之间转移。

卖饮料的自动售货机就是一种有穷自动机：假设其中的饮料价格都是整数元，而且只接收 5 块钱的纸币，根据余额的不同，可能状态有 6 个：5 元、4 元、3 元、2 元、1 元、0 元。你塞进去 5 块钱，此时的状态就是“5 元”，你点了一罐可乐，花去 3 元，于是状态切换到“2 元”，这时候你按下“退币”，就把剩下的 2 元退给你，并把状态切换到“0 元”，“0 元”这个状态也叫作“最终状态”。到此，这一轮状态转移结束，如果你再塞 5 块钱，就开始新一轮的状态转移。

严格说起来，有穷自动机必须满足 4 个条件：

（1）具有有限多个状态；
（2）有一套状态转移函数（或者叫“规则”）；
（3）有一个开始状态；
（4）有一个或多个最终状态。

我们说自动售货机是一种有穷自动机，就是因为它满足这 4 个条件：

---

[1] 本章关于自动机原理的部分示意图来自在 Google 工作的 Russ Cox（http://swtch.com/~rsc/），已经获得作者本人许可。

（1）具有有限多个状态（6 个）；

（2）有一套状态转移函数（比如余额还有 3 元，你买了一罐 2 元的饮料，则转移到状态“1 元”，如果你选择买 4 元的饮料，则报告“余额不足”，状态并不变化）；

（3）有一个开始状态（余额“5 元”）；

（4）有一个最终状态（余额“0 元”）。

自动售货机对应的有穷自动机模型如图 8-1 所示，它包含 6 个状态，对应余额的 6 种可能，起始状态是“￥5”，结束状态是“￥0”，每个箭头代表一个转移函数（每样商品的价格为 1 元或者 2 元，所以每个转移函数上的文字或者是-1，或者是-2），在￥4、￥3、￥2、￥1 状态下，都可以直接退币，所以有一个转移函数直达最终状态。

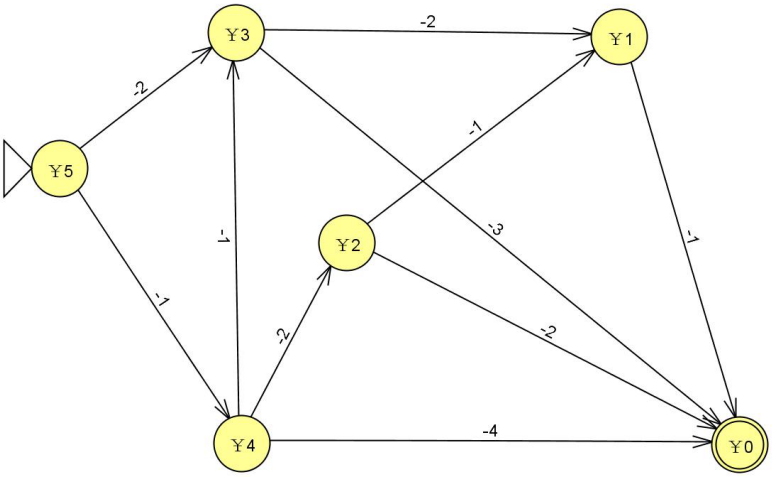


图 8-1 自动售货机对应的有穷自动机模型

自动售货机并不关心你买了什么商品，也不关心你的选择顺序，无论你买什么，它总处在这 6 个状态之一，只需要根据状态转移函数在其中转移即可。

8.2 正则表达式的匹配过程

正则表达式所使用的理论模型就是有穷自动机，其具体实现称为**正则引擎**（Regex Engine）。用正则表达式处理字符串，首先需要生成自动机（你应该还记得，很多语言中使用正则表达式之前都要“编译”正则对象）；之后，无论输入什么字符串，正则引擎都只需要老老实实地在状态之间游走。

图 8-2 显示了正则表达式 `a(bb)+a` 对应的自动机。这台自动机的表示与之前看到的稍有不同：在匹配字符串时，输入的都是字符，所以箭头上标注的都是字符。

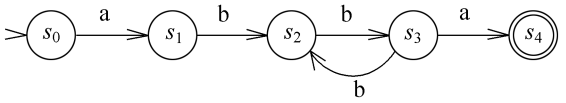


图 8–2　a(bb)+a 对应的有穷自动机

在这台有穷自动机中，S_0、S_1、…、S_4 是各个状态，S_0 为开始状态，S_4 为最终状态；转移函数很直观：比如当前状态是 S_0，输入字符 a，则转移到 S_1；如果当前状态为 S_0，输入的不是 a，那么直接退出。这也很好理解：如果正则表达式是 `a(bb)+a`，它能匹配的字符串只能是以字符 a 开头的，否则必然不能匹配。

图 8-3 说明了这台有穷自动机对字符串 abbbba 的处理过程。

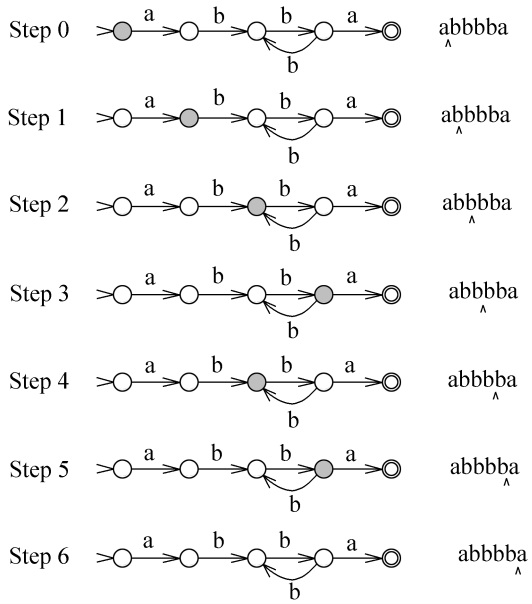


图 8–3　自动机对字符串 abbbba 的匹配过程

在经历了一系列的状态转移之后，字符串 abbbba 处理完毕，自动机停留在最终状态上，也就是说，字符串 abbbba 可以由正则表达式 `a(bb)+a` 匹配。

同一个正则表达式对应有穷自动机不止一台，可以是若干台，这些有穷自动机是等价的。同样是正则表达式 `a(bb)+a`，它对应到图 8-4 所示的两台完全等价的有穷自动机。

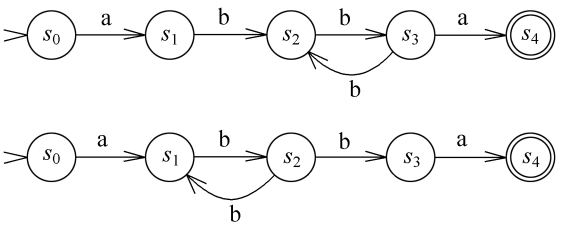


图 8–4　两台完全等价的自动机

仔细观察会发现，第二台自动机有些奇怪，在输入 ab 之后，再输入 b，它所处的状态是不确定的：可能在 S_1，也可能在 S_3。但是，输入 `a(bb)+a` 能匹配的字符串，它确实可以抵达最终状态 S_4。图 8-5 所示的自动机看起来更加奇怪，而且它仍然是与 `a(bb)+a` 完全对应的。

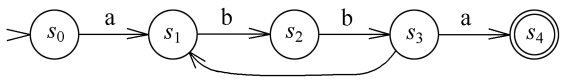


图 8–5 另一台等价的自动机

在状态 S_3，即便没有输入任何字符，也不会停留下来，而可能“凭空”转义到 S_1。也就是说，在某个时刻，自动机到底处在状态 S_3，还是 S_1，这是不确定的！但是这种不确定性，并不会影响自动机对于正则表达式 `a(bb)+a` 的匹配。也就是说，这台有穷自动机与之前的两台有穷自动机，也是完全等价的！

根据状态的确定与否，一般我们会把有穷自动机（正则引擎）分为两类：一类是**确定型有穷自动机**（Definite Finite Automata，简称 DFA），在任何时刻，它所处的状态是确定无疑的；另一类是**非确定型有穷自动机**（Non-definite Finite Automata，简称 NFA），在某个时刻，它所处的状态可能是不确定的。图 8-6 把上面的三台自动机做了分类，第一台是 DFA，而另两台是 NFA。

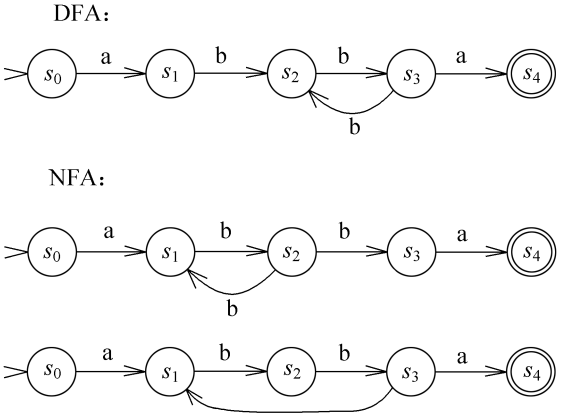


图 8–6 等价的 DFA 和 NFA

可以证明，DFA 和 NFA 之间存在等价关系。也就是说，每一台 DFA 都可以等价转换为一台 NFA，反过来也成立（具体证明过程比较复杂，这里略去，有兴趣的读者可以参考计算理论相关资料）。

比较正则表达式 `a(bb)+a` 和这三台自动机，会发现 NFA 构造起来更直观，实际上这是普遍规律：从正则表达式出发，构造 NFA 的难度要小于 DFA。但是正如之前讲过的，DFA 在任意时刻必定处于某个确定的状态，而 NFA 可能处于若干状态之中的任何一个，所以，如果使用 NFA，就必须保存所有的可能的状态，并且在某种状态不可行时“回退”到之前保存的状态，这就是正则表达式匹配中的重要概念：回溯。

8.3　回溯

比起 DFA，NFA 看起来足够“麻烦”：它的状态是不确定的，这有点像走迷宫，越走岔路口越多，最后不会迷路吗？

不过，NFA 的正则引擎自有办法：如果有多个可能的状态，它们会在选择时记录下这些状态备用，然后才选择其中某个状态尝试；如果之后遇到死路，则退回去，选择最近一次记录的且未尝试过的状态；如果又遇到死路，再选择最近一次记录的且未尝试过的状态……这有点像在分岔路口留下标记——如果我们在遇到的每个分岔路口都留下标记，即便前头是死路，也可以根据标记返回，而不会迷路。

为了说明 NFA 的匹配过程，来看在第 2 章举过的双引号字符串匹配的例子，所用的正则表达式是`".*"`，而字符串是`"quoted string"`，匹配的过程如图 8-7 所示。

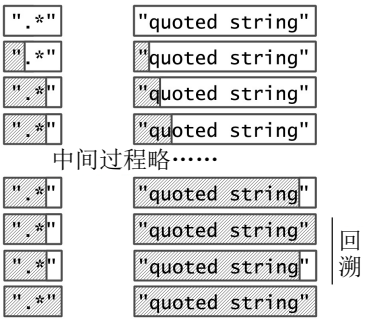


图 8–7　表达式".*"对字符串"quoted string"的匹配过程

从图 8-7 中可以看出，在匹配的过程中，`.*`曾经匹配了 `quoted string`，但为了保证表达式中最后一个`"`的匹配，`.*`不得不“交还”最后的"，这种“尝试失败-重新选择”的过程，就是回溯（backtracking）。

回溯只属于 NFA 引擎。从之前的原理图中可以看到，NFA 匹配时，正则引擎并不准确知道当前的状态，只能在所有状态不确定的地方将各种状态都保存下来（现在已经匹配了哪些字符，进行到字符串中的哪个位置，正则表达式中的哪个位置），逐一尝试，发现此路不通，则退回来，选择最近保存的其他状态尝试……如此持续进行下去，直到达到最终状态（这时候报告“在整个正则表达式开始尝试的位置，匹配成功”）；或者所有可能状态都尝试完毕，仍然不能到达最终状态（如果当前位置是字符串的末尾，则报告“在当前位置匹配失败”；否则，把“整个正则表达式开始尝试的位置”向前推进一个字符，再开始新一轮的尝试）。

看到这里，就不难明白为什么不推荐使用`.*`了，因为`.`几乎能匹配任何字符串（如果明确指定单行模式，则确实能匹配任何字符），而`*`又表示“匹配优先”，所以正则引擎在处理`.*`后的其他元素之前，会先让`.*`“吞掉”几乎整个字符串。仍然是上面的正则表达式，只是字符串变为

"quoted" string，回溯的次数大大增加了，如果在结尾的"之后还有很长的文本，回溯的次数还可能大大增加，匹配过程如图 8-8 所示。

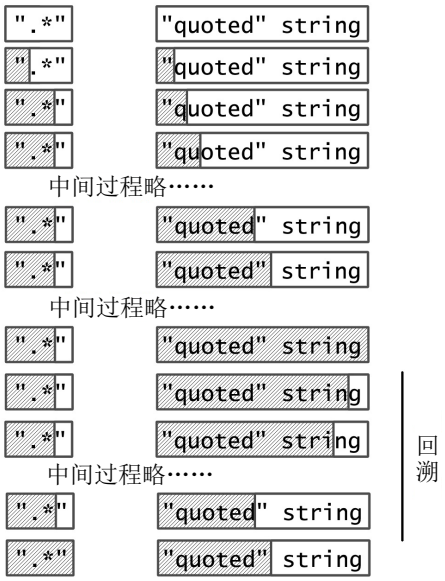


图 8–8 表达式".*"对字符串"quoted" string 的匹配过程

为避免这类问题，最好的办法是准确表达意图，比如规定双引号字符串内部不允许出现双引号字符，就要将表达式改为`"[^"]*"`；当然也可以换用忽略优先量词，将表达式改为`".*?"`，两种办法都可行。[1] 不过总的经验是，除非确实必要，否则尽量不要使用`.*`。

要注意的不仅仅有`.*`，还有更糟的情况，比如`(…*)*`之类的表达式，这时候回溯的次数会呈指数增长，却不会对匹配有任何影响，所以应该绝对避免。第 3 章中匹配 HTML tag 的正则表达式是`<('[^']*'|\"[^\"]*\"|[^'\">])+>`，其中的多选分支`[^'\">]`没有添加量词`*`，就是因为单引号字符串和双引号字符串之外的字符虽然可能有很多，但多选结构最外层还有`+`限定，从忽略之前两个多选分支来看，`([^'\">])+`要好过`([^'\">]*)+`。关于这类问题的详细讲解，可以参考《精通正则表达式》（第 3 版）第 4 章的相关内容。

在实际应用中，不只要注意自己写的正则表达式，还需要防范外界的恶意程序，它们刻意使用会造成大量回溯的表达式，将计算机的资源消耗殆尽，这种攻击有一个专门的名词，叫作正则表达式拒绝服务攻击（Regular Expression Denial of Service）。[2]

---

[1] 其实这个表达式并不够完整，因为有些双引号字符串中是可以出现双引号字符的，比如"\""，其中的双引号字符之前有反斜线，所以它是一个经过转义的双引号。能处理这种情况的表达式在 4.3 节进行了详细讨论。

[2] 详细信息可以参考 http://en.wikipedia.org/wiki/ReDoS。

8.4　NFA 和 DFA

上一节粗略介绍了回溯，它是 NFA 特有的功能，DFA 不需要回溯，也就不需要保存状态，再反复尝试。这样看来，NFA 不是要更慢吗？事实也确实如此，但是当前我们所使用的大多数工具中的正则引擎，都选用了 NFA，这是为什么呢？

NFA 确实更慢，但 NFA 也有自己的优势：如果正则表达式比较复杂，构建 NFA 的时间比 DFA 的时间短（举例来说，如果你的正则表达式使用了多选分支，每个分支其实只是一个简单的字符串，那么完全可以直接对每个多选分支构建简单的 NFA，再把它们简单“并列”起来就可以了；相比之下，构建整个表达式对应的 DFA 就复杂多了）。

同时，现代 NFA 也提供了更多的优化措施，比如之前提到的 `a(bb)+a` 的匹配，优化过的 NFA 可以“并行尝试”，其匹配过程如图 8-9 所示，这样的速度就快多了。

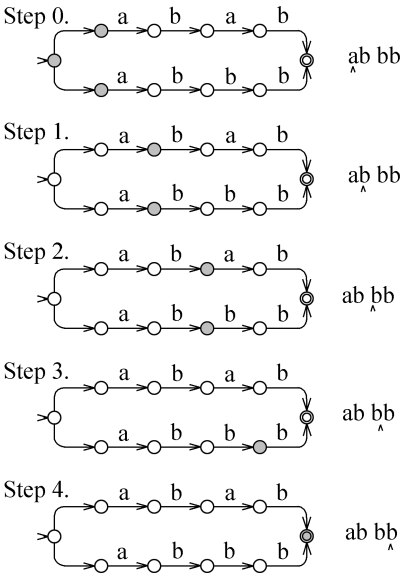


图 8–9　经过优化的 NFA 匹配过程

更重要的是，NFA 的匹配性质决定了它必须在匹配过程中保存可能的状态，需要“停下来四处看看”，所以也能够“回顾一路走来的历程”；相比之下，DFA 不会两次测试同一个字符，所以不需要保存状态。因此，NFA 具有许多 DFA 无法提供的功能：比如捕获型括号 `(…)`，反向引用 `\num`，环视功能 `(?!…)`、`(?=…)`，忽略优先量词 `+?`、`*?`、`??`……

如果希望用到这些功能，一定不要选择使用 DFA 引擎的工具。当然一般来说，用户并不需要操心引擎是 DFA 或者 NFA，毕竟它们是位于“幕后”的，需要关注的是，是否提供了希望实

现功能所用到的 API。

而且，在现代的一些工具中，为兼顾效率和功能，同时包含了 DFA 和 NFA 两种引擎，如果发现正则表达式中没有专属于 NFA 的功能，则使用 DFA，否则使用 NFA。表 8-1 列出了各种常用工具所使用的正则引擎。

表 8–1 常用工具所使用的正则引擎

引擎类型	程序
DFA	awk（大多数版本）、egrep（大多数版本）、flex、lex、MySQL、Procmail、Golang
NFA	GNU Emacs、Java、grep、less、more、.NET、Python、Ruby、PHP、sed、Objective-C

细分起来，NFA 又有传统型 NFA（Traditional NFA）和 POSIX NFA 两种。两者的主要区别在于，如果多选分支中的多个分支都能匹配，传统型 NFA 优先选择左侧的分支，而 POSIX NFA 一定要选择最长的分支。比如用表达式`(jeff|jeffrey)`匹配字符串 `jeffrey`，POSIX NFA 的结果是 `jeffrey`，传统型 NFA 的结果则是 `jeff`——如果调换多选分支的顺序，写成`(jeffrey|jeff)`，POSIX NFA 的结果不变，传统型 NFA 的结果则变为 `jeffrey`。

问题看起来很复杂，具体使用起来其实比较简单：POSIX NFA 的应用很少，主要用于 Linux/UNIX 下的工具（所以它们中的很多并不支持捕获分组，详见第 16 章），本书中介绍的编程语言基本都采用传统型 NFA 引擎。保险起见，不妨这样记忆：一般情况下，多选分支优先采用最左侧的分支。这一点务必要熟记：使用多选结构进行正则表达式操作时，很可能因为多选结构的顺序问题得到不同的结果，具体情况在下一章介绍。

关于正则表达式的匹配原理就介绍这么多，如果对这部分内容有更深入的兴趣，可以参考《精通正则表达式》（第 3 版）的第 4 章。

第 9 章　常见问题的解决思路

前面已经讲解了正则表达式的常用功能和结构，Unicode 问题，正则表达式的匹配原理，关于正则表达式的通用知识已经基本介绍完毕。不过，使用正则表达式是为了解决各种具体问题，所以这一章专门讲解使用正则表达式解决具体问题的思路和注意事项。

9.1　关于元素的三种逻辑

正则表达式提供了一整套描述文本特征的方法，它的匹配其实也就是查找符合所描述特征的文本的过程。按照元素（单个字符、字符组、多选分支等）的出现情况，这些特征分为三类，也可称为三种逻辑：必须出现、可能出现、不能出现，具体解释如表 9-1。

表 9-1　关于元素的三种逻辑

分类	说明
必须出现	某些元素必须出现
可能出现	某个元素或许出现，或许不出现，或许长度不固定；要出现的，是某几个元素中的一个
不能出现	某些元素不能出现

不管正则表达式多么复杂，总是这三种逻辑的组合。比如匹配双引号字符串的任务，可以按三种逻辑分析如下。

必须出现	首尾的双引号字符必须出现
可能出现	两个双引号之间的字符个数是不确定的（如果是空字符串""，则两个双引号之间没有字符）
不能出现	两个双引号之间不允许出现双引号字符

再比如，匹配 HTML 中的 open tag（如<h1>）和 close tag（如</h1>），按三种逻辑分析如下。

必须出现	首尾必须分别是<和>，如果是 close tag，则<之后必须出现/
可能出现	<和>之间的字符串长度必须大于 1（<>不是一个合法的 tag）
不能出现	<和>之间不能有>，如果是 open tag，<之后不能出现/

9.1.1 必须出现

“必须出现”是正则表达式中最普通的逻辑关系，它表示某个元素必须出现。通常，这些元素是所要匹配文本的最重要特征：查找 tag，则必须出现的是字符<和>；查找 E-mail 地址，必须出现的元素是@。

如果某个元素必须出现，通常不会（也不应该）用量词（*、?）来限制，也不出现在多选结构中（如果把普通的字符串也看作正则表达式，那么其中每个字符都是必须出现的）。所以，如果在匹配 tag 的正则表达式中，<和>出现在某个多选结构内，或者之后跟有?、*之类的量词，那么这个表达式多半有问题；同样，匹配 E-mail 地址的正则表达式中的@字符，也不应该有量词限定，或者出现在多选结构内部。

这条要求看起来简单，但是在日常使用中却很容易犯错误。经常遇到的情况是，为了考虑更多的可能情况而修改正则表达式，增加量词和多选结构，最后正则表达式中没有任何必须出现的元素。比如匹配数字字符串的正则表达式，一个数字字符串可能包含三个部分：开头的+或-、整数部分、小数点和小数部分，然后分别列出匹配这三个部分的正则表达式。

符号	`[-+]`
整数部分	`\d+`
小数部分	`\.\d+`

单独来看，这三个部分都不是必须出现的：数字字符串可以没有符号，比如 `3.14`；也可以没有整数部分，比如`.14`；还可以没有小数点和小数部分，比如`-3`。所以，有些人就把对应的表达式元素加上量词，最终得到`[-+]?(\d+)?(\.\d+)?`。初看起来，这样并没有错，仔细看看却发现，这个表达式中所有元素都不是必须出现的，换句话说，这个表达式匹配成功时，可以不匹配任何文本。

简单去掉量词并不能解决问题，这样又会错过许多本应该匹配的文本。真正要做的，是理清表达式各个部分的关系，尤其是“可能出现”和“必须出现”之间的关系。

9.1.2 可能出现

与普通字符串处理相比，“可能出现”可以算作正则表达式最明显的特征，也是最常用的逻辑。虽然它看起来很直观，但细说起来，它其实分为两种情况：从元素外来看，元素可能出现也可能不出现，或者出现次数不确定；从元素内来看，元素可能表现为一种形态，也可能表现为另一种形态。

第一种情况需要用量词。在匹配双引号字符串的正则表达式中，两个双引号字符是必须出现的，它们之间的文本可能出现，也可能不出现，如果出现，长度没有限制，所以用*来限制；在

匹配 tag 的正则表达式中，<和>之内的 tag 名，必须包括至少一个字符，所以用+来限制。

第二种情况需要使用字符组或多选结构。如果各种可能形态都是单个字符，则使用字符组，比如上一节匹配数字字符串正则表达式中的[-+]，它说明“此处可能出现+号，也可能出现-号”。如果某一种可能形态不只是一个字符，比如 this 或 that，则应该使用多选分支(this|that)。

回过头来看数字字符串的匹配，根据上一节的分析，符号部分、整数部分、小数部分（包括小数点）都是可能出现的，但这种“可能”其实是需要细分的。

符号部分的“可能出现”其实是“可能出现也可能不出现的”；整数和小数部分的可能出现情况要复杂一点，需要研究可能出现的几种形态：只出现整数部分；只出现小数部分；整数部分和小数部分都出现。

只出现整数部分	\d+
只出现小数部分	\.\d+
整数部分和小数部分都出现	\d+\.\d+

使用多选结构将这三种形态统一起来，得到(\d+|\.\d+|\d+\.\d+)。最后得到的正则表达式就是[-+]?(\d+|\.\d+|\d+\.\d+)。需要补充的是：前一章讲过，传统型 NFA 的匹配结果与多选分支的顺序有关，它会优先选择最左侧的多选分支。所以如果你用这个表达式来提取（而不是验证）3.2 之类的数字，只能提取出 3，如果要完整提取出 3.2，必须改换多选分支的顺序，将匹配长度最长的分支移到左边，也就是[-+]?(\d+\.\d+|\.\d+|\d+)。这个问题先不多说，在 9.2 节还会详细讲到。

有些人会觉得(\d+|\.\d+|\d\.\d+)麻烦，所以会把多选结构的各个分支合并，得到(\d+)?\.?\d+。这两个正则表达式能匹配的文本的确相同，但我并不推荐这样写正则表达式，因为它的逻辑不够清晰：对表达式(\d+)?\.?\d+来说，最后的\d+是必须出现的。如果文本只包含整数部分，比如 3，\d+匹配的是整数部分，但如果文本包含小数部分，比如 3.14，则\d+匹配的是小数部分的数字。表达式(\d+|\.\d+|\d+\.\d+)虽然麻烦一点，却是一目了然。

这个例子在第 3 章出现过，当时使用的表达式[-+]?(\d+|\.\d+|\d+\.\d+)仍然不够完美，它可能错误匹配-.14。此时，[-+]?中真正匹配的是-，(\d+|\.\d+|\d+\.\d+)中真正匹配的是\.\d+。

要解决这个问题，可以将符号-和+分情况对待：如果是+，则之后的表达式有三种可能：只有整数部分；有整数和小数部分；只有小数部分，所以用表达式(\d+|\.\d+|\d+\.\d+)匹配；如果是-，则之后的表达式只有两种可能，不可能出现“只有小数部分”的情况，所以用表达式(\d+|\d+\.\d+)匹配。

综合这些情况，最终得到的表达式就是(+?(\d+|\.\d+|\d+\.\d+)|-?(\d+|\d+\.\d+))。

9.1.3　不能出现

“不能出现”是正则表达式中最难处理的。

最简单的“不能出现”可以直接使用排除型字符组，它表示“此处必须出现一个字符，但不能是某些字符”。比如匹配双引号字符串，首尾两个双引号之间的字符，都不能出现双引号字符，用`[^"]`表示，上一节说到，这部分长度可以为零，应当使用量词`*`，所以整个表达式就是`"[^"]*"`。

再比如匹配 HTML tag，在<和>之间“不能出现”>，用`[^>]`表示，上一节说到应当使用量词`+`，所以整个表达式就是`<[^>]+>`，它在第 3 章也出现过，这个表达式仍然可能错误匹配</>。为解决这个问题，还需要细分两种可能：如果<之后是/，则还必须出现至少一个不为>的字符，比如，应当使用表达式`</[^>]+`；如果<之后出现的字符不是/，则在这个字符之后、>之前，还可能出现>之外的字符，并且可能不出现，如果出现，长度没有限制，比如在<a>中，在`[^>]`匹配之后、>之前，就没有任何字符了，所以应当使用表达式`[^/][^>]*`。

不过，这只是最简单的情况，也就是“不允许出现的是字符”；更复杂的情况是，不允许出现某个字符串，比如 E-mail 地址中的用户名（username）就是如此。

在第 3 章曾经讲过电子邮件地址的匹配（☞ 38），使用`[-\w.]{0,64}`匹配邮件地址中的用户名，但是如果仔细阅读E-mail的规范 [1]就会发现，点号和横线都不能出现在用户名的开头，也不能连续出现，也就是说，`-user`、`.user`和`John..Doe`都不是合法的电子邮件地址。

点号和横线不能出现在开头的情况比较好满足，可以用`\w`匹配“非点号非横线字符”，既然整个用户名不能超过 64 个字符，那么之后的字符串长度不超过 63 个字符。综合起来，得到`[\w][\w.]{0,63}`。可是，不允许出现连续两个点号的要求则很难满足，所以下面集中讨论“不超过 63 个字符”部分的匹配。

要求不能出现两个连续点号，许多人的直观反应是`[^.][^.]`，所以整个表达式改为`[\w.]{0,63}[^.][^.][\w.]{0,63}`。但是这样行不通，原因有两点：首先，在`[^.][^.]`前后的两个`[\w.]{0,63}`能匹配的文本，总长度其实在 0~126 之间，无论我们如何修改量词，也不能在两个量词之间建立联系，保证总长度在 0~63 之内；其次也是最重要的，`[^.][^.]`的真正意思是“找到连续的两个非点号字符”，正则表达式匹配时会尽力满足这一要求，即便是类似 `123..456` 这样明显包含两个连续点号的文本，`[^.][^.]`仍然可以在其中找到四处匹配：`12`、`23`、`45`、`56`，同时左右两侧的`[-\w.]{0,63}`仍然可以成功匹配。

还有人想到的是效仿排除型字符组，用`(^\.\.)`来表示“不能出现两个连续点号”，遗憾的是，这样也行不通，因为正则表达式只有排除型字符组，没有“排除型括号”。

那么，不妨反过来思考：不能出现两个连续点号，意思是这段文本中的所有字符的右侧都不能是两个连续点号，用环视表示就是`[-\w.](?!\.\.)`，这样的字符最多有 63 个。所以，就得

[1] 请参考 http://en.wikipedia.org/wiki/Email_address，虽然根据规范，username 中还可以包含! # $ % * / ? ^ ` { | } ~等字符，但在实际应用中，许多邮件服务提供商只支持字母、数字、下画线、横线、点号，所以暂不考虑其他字符。

到了表达式`([-\w.](?!\.\.)){0,63}`。请注意，因为`[-\w.](?!\.\.)`不是单个字符也不是字符组，所以用量词限定时，必须使用括号将整个子表达式分为一组；再在表达式的开头加上之前提到的`\w`，保证第一个字符的正确性。从例 9-1 可以看到，这个表达式完全可以保证字符串不会以点号开头，不会包含连续点号，也不会超出规定长度。

例 9–1　使用环视实现“不能出现”的逻辑

```
#注意验证时要在首尾加上\A 和\Z
usernameRegex = r"\A\w([-\w.](?!\.\.)){0,63}\Z"
#合法的用户名
re.search(usernameRegex, "abc123_") != None       # => True
re.search(usernameRegex, "abc1-2.3_") != None     # => True
#包含连续点号
re.search(usernameRegex, "abc1-2..3_") != None    # => False
#开头字符不合法
re.search(usernameRegex, ".abc1-2.3_") != None    # => False
re.search(usernameRegex, "-abc1-2.3") != None     # => False
#长度超过限制
re.search(usernameRegex, "0"*65) != None          # => False
```

也可以更进一步，把“第一个字符不能是点号或横线”也用环视表达，整个表达式就是`(?![-.])([-\w.](?!\.\.)){1,64}`。实际上，这个表达式确实更直观、更容易理解，只是注意量词的下限应当改为 1，因为用户名不能是空字符串。

总结一下：正则表达式中的“不能匹配”，最简单的情况可以用排除型字符组直接表示，但它只能表示“某个字符不能出现”；如果要表示“某个字符串不能出现”，一般都要用到否定环视，其逻辑是：在文本中的每个位置，都用环视否定“不能出现”的字符串（除此之外，还有另一种逻辑，下面讲解验证操作时会看到）。不过，一些比较古老的工具（比如 Apache 1.3，以及 Linux/UNIX 下的某些工具）并不支持否定环视，所以使用时必须留意。

9.2　正则表达式的常见操作

正则表达式执行的操作，可以粗略分为三大类：**匹配、替换、切分**。其中，匹配是最基本的操作——使用正则表达式无非是“用正则表达式匹配文本”，这种说法没错，但太笼统，细究起来，广义的“匹配”又可以分为两类：**提取**和**验证**。所以此处将常见操作分为四类，逐一详细讲解。

9.2.1　提取

数据提取，就是“用正则表达式遍历整个字符串，找出能够匹配的文本”，它主要用来提取所需要的数据，常见的任务有：找出文本中的电子邮件地址，找出 HTML 代码中的图片、超链

接……提取数据时要注意三点：**完整、精确、效率**。

首先来看完整和精确，它们关系很紧密，所以集中讲解（见图 9-1）。完整是要保证正则表达式能匹配足够多的文本，**兼容可能出现的各种情况**，避免疏漏；精确则是要保证正则表达式匹配足够少的文本，**避免错误匹配其他文本**。

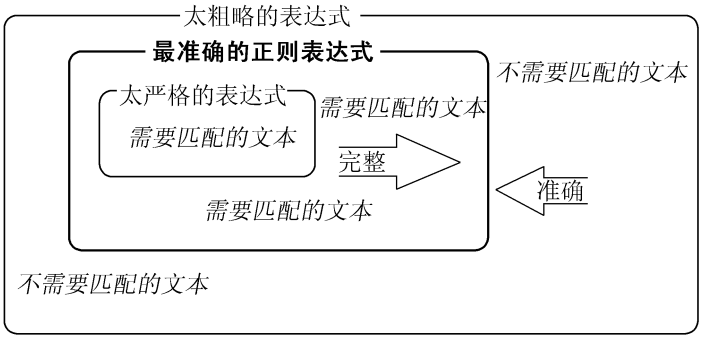


图 9-1 完整和精确

仍然举之前匹配浮点数的例子，回顾逐步完善这个表达式的过程。

（1）常见的浮点数字符串就是 `3.24`、`0.618` 之类，也就是“数字字符串 + 小数点 + 数字字符串”，所以用表达式`\d+\.\d+`。

（2）它无法匹配只有小数点和小数部分的浮点数，比如`.618`，所以应当把整数部分改为可能出现的，得到`\d*\.\d+`。

（3）它无法匹配只有整数部分的浮点数，比如 `3`，所以应当把小数点和小数部分改为可能出现的，得到`\d*\.?\d*`。

（4）表达式中的小数点和小数部分是一体的，要么同时出现，要么同时不出现，所以应当用括号分组功能，得到`\d*(\.\d)*`。

（5）表达式中，所有元素都有量词限制，可以不匹配任何字符，所以它可以匹配空字符串，或者匹配单独的点号，所以应当分情况考虑，得到`(\d+|\.\d+|\d+\.\d+)`。

（6）表达式还不能匹配包含符号的浮点数，比如`+0.8`、`-0.7` 之类，所以还应当加上对符号的处理，得到`[-+]?(\d+|\.\d+|\d+\.\d+)`。

（7）这个表达式可能匹配`-.7`，但通常它并不是一个合法的浮点数字符串，而应当写作-0.7，所以应当再次修改，最后得到的表达式就是`(+?(\d+|\.\d+|\d+\.\d+)|-?(\d+|\d+\.\d+))`。

其中第 2 步、第 3 步、第 6 步，都是为了兼容更多情况而做的修改，也就是保证匹配的完整性。但是，这种修改可能有副作用，影响整个表达式的匹配，所以第 4 步、第 5 步、第 7 步是给这些修改打上补丁，保证准确性，避免错误匹配其他文本。

不过，这个表达式还不够精确——因为提取操作往往是在大段文本中进行的，真正感兴趣的文本并不是独立存在的——如果遇到日期字符串 2010.12.22，就会错误提取出 2010.12；同样，如果用表达式`\d{0,6}`提取邮政编码，遇到手机号码 13801234567，也可能错误提取出 138012。

为解决这个问题，保证精确性，需要**为正则表达式设定合适的断言**。针对浮点数的匹配，为了避免日期字符串的错误匹配，可以在表达式首尾加上否定环视`(?!<\.)`和`(?!\.)`，保证匹配文本的两端不能为点号，于是得到正则表达式`(?!<\.)(+?(\d+|\.\d+|\d+\.\d+)|-?(\d+|\d+\.\d+))(?!\.)`。这样，就真正避免了 2010.12.22 的错误匹配。

提取数据是正则表达式常见的操作，而完整和精确又是非常重要的方面，所以值得多花点篇幅，下面再举一个匹配双引号字符串的例子。

前面用来匹配双引号字符串的正则表达式`"[^"]*"`其实并不够完整——因为双引号字符串内部可能还嵌套转义的双引号。如果真的出现这样的双引号，就可能出现意外：

字符串

```
"quoted string with \"nested quote\""
```

匹配结果

```
"quoted string with \"
\""
```

因为字符串中出现了两个转义序列\"，以`"[^"]*"`来匹配就会分开匹配两次。

改进这个正则表达式，最容易想到的是使用逆序环视，约定"之前必须出现反斜线字符\，所以`"[^"]*"`要改为`"((?<=\\)"|[^"])*"`，因为`(?<=\\)"`能匹配字符串中的\"，所以"quoted string with \"nested quote\""就可以被完整匹配。

现在，完整性的问题已经解决。下面来看精确性，尝试用这个表达式匹配更复杂的文本，看看是否会错误匹配不应该匹配的文本。

字符串

```
"escaped backslash\\" and "another"
```

匹配结果

```
"escaped backslash\\" and "
```

字符串中只出现了一个转义序列\\，表示反斜线字符。因为`((?<=\\)"|[^"])`中的`(?<=\\)"`匹配字符串中的\"，没有准确识别字符串中的转义序列\\。

看来应该修改逆序环视：`((?<=\\)"|[^"])`匹配的\"之前不能再出现\，也就是说，"之前应

当出现\，但不能出现\\，所以需要用嵌套环视把表达式改为"((?<!\\\\)(?<=\\)"|[^"])*"；这样，就解决了"escaped backslash\\" and "匹配的问题。

不过还有更复杂的情况，比如"double escaped backslash\\\""是完整的双引号字符串，其中出现了相连的转义序列：\\\"，而(?<!\\\\)(?<=\\)"要求"之前不能出现两个连续的\，所以会出错。解决思路之一是，约定"之前出现的\的数目为奇数，可是除去.NET，大多数语言中的逆序环视都要求表达式匹配的文本长度是固定的，所以这个办法并不实用。那么，有没有通用的解法呢？

如果从处理一般的转义序列出发，用\\.匹配转义序列，就可以一次匹配整个转义序列，而不只是匹配转义中\之后的字符，把表达式改为"(\\.|[^"])*"。这时候，既可以完整匹配"escaped backslash\\\"" and "another"，也可以匹配"escaped backslash\\" and "another"中的"escaped backslash\\"和"another"。

现在完整性的问题基本解决，为确保精确性还必须确认不会匹配错误的字符串。比如字符串"quoted string without ending \"，除去开头的双引号和转义的双引号，没有结尾的双引号，所以不是一个合法的双引号字符串，也就不应该能匹配，但可以被"(\\.|[^"])*"错误匹配。

这是为什么呢？来分析匹配的过程：字符串结尾的\"，本来应该由多选结构中的\\.匹配，这时候字符串中已经没有剩下字符了，而表达式结尾的"还没有匹配，所以需要回溯。这时候，\\.“交还”的\"中，\可以由[^"]匹配，还剩下"，正则表达式中也剩下"没有匹配，所以匹配完成——尽管结果是错的。

问题的症结在于，多选结构(\\.|[^"])中，\\.交还的部分可能被[^"]匹配，而我们本意是希望这两个多选分支应对不同的情况，前者匹配转义序列，后者匹配非转义序列。但是仔细想想就能知道，在双引号字符串中的非转义序列中，不但不能出现"，也不能出现\。讲解多选结构时曾经提到：应当尽可能避免多选分支匹配相同的文本，所以应当把多选结构改为(\\.|[^\\"])，整个表达式改为"(\\.|[^\\"])*"。

最后得到的正则表达式才算是真正保证了完整性——可以匹配包含转义双引号的双引号字符串，也保证了精确性——包含转义双引号，但没有结束双引号的非法双引号字符串，是不能匹配的。

关于完整性和精确性，再看最后一个例子。第 3 章提到用正则表达式提取超链接，当时使用的正则表达式是<a\s+href\s*=\s*[\"']?([^\"'\s]+)[\"']?>([^<]+)，其中([^\"'\s]+)用来匹配超链接的地址，([^<]+)用来匹配超链接的文本。但是，这个表达式其实只能匹配text这类“标准”的超链接。

然而在<a 和 href 之间，可能还存在其他内容，比如<a id="mh_img" href…。为了应付这样的情况，似乎可以将<a\s+href 换成<a\s[^>]*\s?href，这样既能匹配<a href…，又能匹配]*\s?中的[^>]和\s 都不是

必须出现的，所以整个表达式也可以匹配<a nonhref…。

仔细想想，<a 和 href 之间出现其他内容，可以由表达式[^>]*匹配；同时要保证<a 之后、href 之前都出现空白字符，最适合使用可以进行“原地判断”的环视，<a(?=\s)和(?<=\s)href匹配了符合条件的<a 和 href，同时省去了必须出现两个\s 的限制，所以匹配这一段的表达式就是<a(?=\s)[^>]*(?<=\s)href。

再看 href 属性之后的部分，之前的表达式中用[\"']?>来匹配，但也不够完整，因为其中还可能有其他内容，比如，不过这种情况处理起来很简单，用[^>]*就可以处理，所以[\"']?>要改为[\"']?[^>]*>。

最后考虑<a>和之间的内容，之前用([^<]+)匹配（添加括号分组是为了提取出这段内容），是考虑这里的内容不包含其他 tag，比如text，但实际情况并非如此，其中可能是图片。因为这段内容可能包括任何字符，甚至包括换行符，所以最好用[\s\S]（如果指定了单行模式，也可以使用点号）匹配其中的字符，比较好的办法是使用忽略优先量词，以([\s\S]+?)匹配，忽略优先量词保证了这个表达式的匹配到之前结束。

综合起来，兼顾完整和精确的匹配超链接的表达式就是<a(?=\s)[^>]*(?<=\s)href\s*=\s*[\"']?([^\"'\s]+)[\"']?[^>]*>([\s\S]+?)。

完整和精确很重要，也很费心思；但也有很多时候不用考虑这么细致，**这取决于你要处理的文本**。如果要处理的文本中只会出现像 3.14 一样规则的浮点数，完全可以放心使用\d+.\d+；如果确认文本中不会出现手机号码之类的字符串，完全可以放心使用\d{0,6}；如果要处理的都是简单的双引号字符串，完全可以放心使用"[^"]*"……

提取数据时，应当注意的另一点是效率。在讲解正则表达式的匹配原理时已经提到过，在编写表达式时需要注意效率，这里要说的是另一方面：提取的效率不仅与正则表达式本身有关，也与调用的 API 有关。许多网站每天的访问日志经常有几百兆甚至更大，如果先读入整个日志文件，再一次性提取某些数据（比如访问者 IP），就可能消耗非常长的时间。

遇到这种情况，可取的办法是在条件允许的情况下（比如只需要逐步提取出来，依次处理），逐步进行提取——提取一个结果，处理，再提取一个结果，再处理……表 9-2 列出了常用语言中的提取 API。

表 9–2　常用语言中的提取 API 方法

语言	方法	备注
.NET	Regex.**match**(*string*)	逐次进行
	Regex.**matches**(*string*, *regex*)	一次性进行

（续表）

语言	方法	备注
Java	`Matcher.find()`	逐步进行
JavaScript	`RegExp.exec(string)`	一次性进行
	`string.match(RegExp)`	一次性进行
PHP	`preg_match(pattern, subject, matches)`	逐步进行
	`preg_match_all(pattern, subject, matches)`	一次性进行
Python	`re.find(pattern, string)`	逐步进行
	`re.finditer(pattern, string)`	逐步进行
Ruby	`Regexp.match(str)`	只能找到第一次匹配
	`string.index(Regexp, int)`	逐步进行
	`string.scan(Regexp)`	一次性进行

一次性提取所有匹配结果的操作不必说太多；这里只讲解“逐步进行”时如何真正保证“逐步”？也就是说，在第二次调用匹配时，如何保证是“承接”第一次调用，找到下一个匹配结果。下面介绍三种常用的办法，例子是查找字符串 `123 45 6` 中的数字字符串，希望依次输出 `123`、`45`、`6`。

如果使用函数式正则表达式采用的是面向对象式处理，匹配结果对象可以自动保存匹配状态信息，其中包括匹配本次匹配结束的位置，所以下次匹配时，会自动从之前记录的结束位置开始。Java 就是这样，从例 9-2 可以看到，循环调用 `Matcher.find()`方法，就可以逐个获得所有匹配（在.NET 中，是循环调用 `Match.NextMatch()`）。

例 9–2 面向对象式处理中的逐次提取(Java)

```
String str = "123 45 6";
Pattern p = Pattern.compile("\\d+");
Matcher m = p.matcher(str);
while (m.find()) {
 System.out.println(m.group());
}
```

如果使用函数式处理，则必须自己保存匹配的状态信息，手动指定偏移值。多数语言都有办法在匹配时指定偏移值，也就是“从字符串的*offset*位置开始尝试匹配”。如果要逐一获得所有匹配，每次匹配时手动记录上次匹配的结束位置，下次匹配时，指定从此位置开始尝试即可。[1] 在

[1] 注意，字符串处理时可能有人习惯将偏移值指定为“上一次匹配的起始位置+1”，但正则表达式处理时这样是不对的，比如正则表达式是`\d+`，而字符串是"123 45 6"，第一次匹配的结果是 123，如果把偏移值设定为“上一次匹配的起始位置+1”，之后的匹配结果就是 23，3……

PHP、JavaScript、Ruby、Python中，通常采用这种办法，例 9-3 给出了PHP的代码，请注意其中的*lastOffset*变量。

例 9–3　面向对象式处理中的逐次提取(PHP)

```
$string="123 45 6";
$regex="/\\d+/";
$matched=1;
$oneMatch=array();
$lastOffset=0;
$matched = preg_match($regex, $string, $oneMatch, PREG_OFFSET_CAPTURE, $lastOffset);
while ($matched == 1) {
    //上次匹配开始位置+上次匹配文本的长度，得到上次匹配的结束位置
    $lastOffset = $oneMatch[0][1]  + strlen($oneMatch[0][0]);
    echo $oneMatch[0][0]."<br />";
   $matched = preg_match($regex, $string, $oneMatch, PREG_OFFSET_CAPTURE, $lastOffset);
}
```

最后一种办法是使用迭代器。例 9-4 展示了在 Python 中，调用 `re.finditer(`*`pattern`*`,`*`string`*`)`，会得到一个迭代器（虽然看起来是用 for 循环，但每次的匹配结果其实是临时计算得到的），这种办法目前只有 Python 提供。

例 9–4　使用迭代器进行逐次提取(Python)

```
for match in re.finditer(r"\d+", "123 45 6"):
    print match.group(0),
```

提取操作还有一点容易被忽视，就是多选分支的顺序。因为大多数语言和工具都使用传统型 NFA 引擎（详见第 6 章），多个匹配分支都能匹配时，优先选择左侧的分支。所以如果多个分支的匹配存在重叠，提取时一定要把能匹配更长文本的分支放在左侧。比如正则表达式`\d+|\d+\.\d+`，按照预期，`\d+`匹配 `24` 之类的数字，`\d+\.\d+`匹配 `3.14` 之类的数字，如果字符串是 `2.72`，应当可以全部提取出来，但是因为`\d+`在左侧，优先得到匹配，所以提取的结果是 `2`，而不是 `2.72`。

9.2.2　验证

数据验证是另一类“匹配”操作，它用来“检查字符串能否**完全由正则表达式匹配**”，主要用来测试和保证数据的合法性，比如 Web 编程中经常用正则表达式验证用户输入数据，比如手机号码、邮件地址、QQ 号码等。

初看起来，这也是用正则表达式在字符串中查找匹配文本。仔细分析又不一样：提取返回的是字符串，验证返回的是布尔值；提取时，正则表达式最终匹配的起始/结束位置都是不确定的，需要逐次试错才能确定；验证时，正则表达式的起始/结束位置必须定位于字符串的起始/结束位置，从字符串起始位置开始尝试匹配，只要表达式中任意一个必须匹配的元素出现无法匹配的情

况，即宣告验证失败，不必逐次试错。假设正则表达式是`\d{6}`，字符串是 `a123456b`，很明显，提取操作中应当返回 `123456`，但验证操作应当返回 False，因为表达式匹配的并不是整个字符串。

针对验证操作的特点，有些语言中提供了专门的方法进行正则表达式验证，它们保证正则表达式匹配的起始/结束位置就是字符串的起始/结束位置，而且返回布尔值。如果没有提供专门的方法，通常的办法是手动在正则表达式的首尾加上匹配字符串起始/结束位置的`\A`和`\z`，再通过返回值判断。[1]

这里，有必要单独谈谈`\A`和`\z`的问题。许多示范代码（包括本书开头部分）都使用`^`和`$`匹配字符串的起始/结束位置，但第 64 页讲过，`$`匹配的其实是“结尾换行符之前的位置”，如果字符串的结尾是换行符`\n`，即便在表达式末尾添加了`$`，匹配时仍然会忽略最后的换行符。所以应当使用真正匹配“结束位置”的锚点`\z`（Java、JavaScript、.NET、Ruby、PHP 都支持`\z`，只有 Python 不支持，Python 中的`\Z`等于其他语言中的`\z`）。虽然`^`和`\A`都是匹配字符串的起始位置，但为了与`\z`对应，推荐使用`\A`。而且使用`\A`和`\z`还有一个好处：如果设定了多行模式，`^`和`$`的匹配会有所不同，`\A`和`\z`则不受影响。表 9-3 中列出了常用语言中的验证方法。

表 9–3 常用语言中的验证方法

语言	验证方法	备注
.NET	`Regex.IsMatch(`*string*`, `*regex*`)`	返回布尔值，但需要`\A`和`\z`
Java	`String.matches(`*regex*`)`	专用于验证，返回布尔值，不需要`\A`和`\z`
JavaScript	`RegExp.test(`*string*`)`	返回布尔值，需要`\A`和`\z`
PHP	`preg_match(`*pattern*`, `*subject*`)!=0`	需要`\A`和`\z`
Python	`re.search(`*pattern*`, `*string*`)!=None`	需要`\A`和`\Z`
	`re.match(`*pattern*`, `*string*`)!=None`	不需要`\A`，只需要`\Z`
Ruby	`Regexp.match(`*string*`)!=nil`	需要`\A`和`\z`
	`String.index(`*Regexp*`)!=nil`	需要`\A`和`\z`

除去方法上的差别，验证与匹配在正则表达式编写上也有区别。因为验证时文本的起始/结束位置是预先知道的，所以验证的表达式编写起来更加简单。比如之前匹配浮点数的表达式，首先得到的正则表达式是`(+?(\d+|\.\d+|\d+\.\d+)|-?(\d+|\d+\.\d+))`，进行数据提取还需要在两端加上环视，防止错误匹配其他字符；但是如果是数据验证，就可以完全忽略考虑两端的环视，应该/不应该出现什么字符，直接在首尾加上`\A`和`\z`即可，最终，验证浮点数的表达式是`\A(+?(\d+|\.\d+|\d+\.\d+)|-?(\d+|\d+\.\d+))\z`。同理，匹配邮政编码的正则表达式是`\d{6}`，如果用于数据验证，可以直接使用`\A\d{6}\z`。

[1] 专用于验证的方法应当只返回布尔值，所以表 9-3 中对原本不返回布尔值的函数进行了包装，以返回布尔值。

其实，根据这两点差别，完全可以为验证操作找到不同的思路：从根本上说，数据提取的结果是字符串，而验证的结果是布尔值。既然返回布尔值，就很容易想到布尔运算，所以验证时不妨简单将各个条件叠加起来，直接得到最后的表达式。下面以密码字符串的验证为例，说明这种思路。

用正则表达式验证密码字符串是否合法，“合法”密码必须同时满足以下所有要求。

（1）密码长度在 6~12 个字符之间；
（2）只能由小写字母、阿拉伯数字、横线组成；
（3）开头和结尾不能是横线；
（4）不能全部是数字；
（5）不允许有连续（2 个及以上）的横线。

针对这 5 点要求，可以分别列出对应的表达式。

（1）密码长度在 6~12 个字符之间：其形式类似`.{6,12}`；
（2）只能由小写字母、阿拉伯数字、横线组成：所有的字符都只能由`[0-9a-z-]`匹配；
（3）开头和结尾不能是横线：开头`\A(?!-)`，结尾`(?<!-)\z`；
（4）不能全部是数字：也就是说，必须出现一个`[^0-9]`或者`\D`；
（5）不允许有连续（2 个及以上）的横线：也就是说，不能出现`--`。

如果用来提取数据，就必须使用能兼顾这 5 点要求的表达式，也就是把上面 5 个表达式综合起来。前三点比较好处理，可以合并为`\A(?!-)[0-9a-z-]{6,12}(?<!-)\z`，剩下的两点要求就麻烦多了。如果你仔细阅读过前面关于三种逻辑的讲解，或许知道能用环视结构解决。但是在验证中完全可以用更简单的办法，只依靠环视解决所有的问题。

既然验证只需要返回布尔值，不妨换个思路，针对每个要求分别判断，再将各个要求的布尔值取“与”（and）运算即可。取得布尔值，最容易想到的就是环视功能，所以下面列出针对 5 点要求的环视。其中值得注意的是第 5 点“不能出现”的处理，之前我们用“每个位置都不能出现”的逻辑来满足“不能出现”的要求，但是此处用到了另一种逻辑：先用子表达式`\A.*--`匹配“不应该出现”的文本，再用否定环视来杜绝这种情况。

（1）密码长度在 6~12 个字符之间：`(?=\A.{6, 12}\z)`；
（2）只能由小写字母、阿拉伯数字、横线组成：`(?=\A[0-9a-z-]*\z)`；
（3）开头和结尾不能是横线：开头`(?!\A-)`，结尾`(?!.*-\z)`；
（4）不能全部是数字：`(?=\A.*[^0-9])`（这里不需要出现`\z`，只要出现了非数字字符就可以）；
（5）不允许有连续（2 个及以上）的横线：`(?!\A.*--)`（这里不需要出现`\z`）。

第 4 章提到过，环视是在原地进行的，而且可以组合起来。而根据环视的组合规则，简单并列多个环视的意思是：找到这样一个开始位置，在这个位置往后，各个环视的判断都必须成功。不妨直接选择字符串的起始位置作为“当前位置”，用`\A`匹配，再将所有环视结构列在后面即可。例 9-5 说明，这个表达式确实可以满足要求。

例 9-5 纯粹使用环视完成验证

```
    #Python 不支持\z，所以\z 必须写成\Z
    passwdRegex = r"\A(?=\A.{6,12}\Z)(?=\A[0-9a-z-]*\Z)(?!\A-)(?!.*-\Z)(?=\A.*[^0-9])
(?!\A.*--)"
    #合法
    re.search(passwdRegex, "abc123-d") != None            # => True
    #超长
    re.search(passwdRegex, "abcdefg1234567890") != None   # => False
    #出现了其他字符
    re.search(passwdRegex, "1234ab*") != None             # => False
    #开头或结尾出现了横线
    re.search(passwdRegex, "abc123-") != None             # => False
    re.search(passwdRegex, "-abc123") != None             # => False
    #出现了连续横线
    re.search(passwdRegex, "abc--123") != None            # => False
```

如果你觉得这个表达式有点复杂，可以对它做一些整理：开头的`\A`定位在字符串的起始位置，而环视不会真正改变匹配的位置，每个环视判断完成，下一个环视仍然从字符串的起始位置开始，因此可以将各个环视中的`\A`去掉，但是末尾的`\z`不能去掉，否则`(?=.{6,12})`就不能保证整个字符串的长度在 6~12 之间。这个表达式的测试结果见例 9-6。

例 9-6 用修改后的表达式完成验证

```
    #Python 不支持\z，所以\z 必须写成\Z
    passwdRegex = r"\A(?=.{6,12}\Z)(?=[0-9a-z-]*\Z)(?!-)(?!.*-\Z)(?=.*[^0-9])
(?!.*--)"
    #合法
    re.search(passwdRegex, "abc123-d") != None            # => True
    #超长
    re.search(passwdRegex, "abcdefg1234567890") != None   # => False
    #出现了其他字符
    re.search(passwdRegex, "1234ab*") != None             # => False
    #开头或结尾出现了横线
    re.search(passwdRegex, "abc123-") != None             # => False
    re.search(passwdRegex, "-abc123") != None             # => False
    #出现了连续横线
    re.search(passwdRegex, "abc--123") != None            # => False
```

这种思路简单易行，对于数据验证特别有用，因为验证操作一般会明确给出验证规则（比如这个例子中的 5 点要求）。相比之下，提取操作不一定会给出关于要提取文本的严格描述，所以一般处理是先找到最明显特征，再逐步完善表达式。

需要补充的是：提取操作时应当注意多选分支的顺序，否则会导致匹配不完整的情况，验证时则**不需要考虑**这一点。仍然举正则表达式`\d+|\d+\.\d+`，字符串 `3.14` 为例，虽然`\d+`可以匹配 `3`，但最终验证的是 `3.14` 是不是能够由表达式完整匹配，这个判断不受多选分支顺序的影响。

9.2.3 替换

替换操作也是一种“匹配”，它通常需要三个参数，其中 *regex*（或 *pattern*）表示查找需要替换文本的正则表达式，*replacement* 表示“要替换成”的字符串，*subject*（或 *string*、*input*）则是要进行替换处理的文本。表 9-4 中列出了常用语言中的替换方法。

表 9–4　常用语言中的替换方法

语言	方法	备注
.NET	`Regex.Replace(`*input*, *pattern*, *replacement*`)`	静态方法
	`Regex.Replace(`*input*, *replacement*`)`	`Regex` 实例事先生成
Java	`String.replaceAll(`*regex*, *replacement*`)`	静态方法
	`String.replaceFirst(`*regex*, *replacement*`)`	同上，但只进行第一次替换
	`Matcher.replaceAll(`*replacement*`)`	`Matcher` 实例事先根据 `Pattern` 和 `String` 生成
	`Matcher.replaceFirst(`*replacement*`)`	同上，但只进行第一次替换
JavaScript	`string.replace(`*RegExp*, *replacement*`)`	*replacement* 为普通函数
	`string.replace(`*RegExp*, *replacement*`)`	*replacement* 为回调函数
PHP	`preg_replace(`*pattern*, *replacement*, *subject*`)`	
	`preg_replace_callback(`*pattern*, *callback*, *subject*`)`	*callback* 为回调函数
Python	`re.sub(`*pattern*, *replacement*, *string*`)`	
	`re.sub(`*pattern*, *replacement*, *string*`)`	*replacement* 为回调函数
Ruby	`String.sub(`*Regexp*, *String*`)`	只替换第一次匹配
	`String.gsub(`*Regexp*, *String*`)`	替换所有匹配

注：以上列出的都是最基本的形式，在默认情况下，替换是对所有的匹配实行的，也就是说，匹配发生了 *n* 次，替换就会发生 *n* 次，但替换发生的最大次数往往可以用一个可选参数指定，如果这个参数设定为负数，则会对所有匹配进行替换。具体细节此处不展开，请参考各语言的详细章节。

最简单的替换是删除不需要的文本，也就是将 *replacement* 设定为空字符串`""`，此时 *regex* 将能匹配的所有文本替换为空字符串，也就是“删除”了。比如将正则表达式`\s+`能匹配的所有文本都替换为空字符串，就删除了所有的空白字符（如果要处理的文本中包含 Unicode 空白字符，比如中文的全角空格，推荐使用 Unicode Property`\p{Z}`（☞ 130）。当然有时候情况没有这么简单，第 73 页的正则表达式可以处理中英文混排的文本，只删除中文文本中的空白字符，而保留英文文本中的空白字符。

替换也可以用来调整数据的格式。比如日期字符串的整理，如 2010 年 12 月 22 日，按照

MM/DD/YYYY 格式就是 12/22/2010，按照 DD/MM/YYYY 格式就是 22/12/2010，按照 YYYY-MM-DD 格式就是 2010-12-22。使用正则表达式替换很容易进行这种调整，具体的例子在第 44 页已经讲解过，这里不再重复。

关于这类替换，唯一要注意的是，如果在 *replacement* 中出现了对捕获分组的引用，同样需要转义，因为 *replacement* 是字符串变量，而\1、\2 之类是正则文字中的记法，不是字符串中的记法，代码如例 9-7 所示。

例 9–7 replacement 中的引用也需要转义

```
print re.sub(r"(\d{4})-(\d{2})-(\d{2})", r"\2/\3/\1", "2010-12-22")
12/22/2010
print re.sub(r"(\d{4})-(\d{2})-(\d{2})", "\\2/\\3/\\1", "2010-12-22")
12/22/2010
#因为\2、\3、\1 表示 ASCII 码值为 2、3、1 的字符，无法显示，所以用 repr()展示
print repr(re.sub(r"(\d{4})-(\d{2})-(\d{2})", "\2/\3/\1", "2010-12-22"))
'\x02/\x03/\x01'
```

表 9-5 和表 9-6 分别总结了常用语言中对普通分组和命名分组的引用记法。

表 9–5 常用语言中对分组的引用

语言	表达式中的反向引用	替换中的反向引用
.NET	\num	$num
Java	\num	$num
JavaScript	\num	$num
PHP	\num	\num 或$num（PHP 4.0.4 以上版本）
Python	\num	\num
Ruby	\num	\num

表 9–6 常用语言中对命名分组的引用

语言	分组记法	表达式中的引用记法	替换时的引用记法
.NET	(?<name>…)	\k<name>	${name}
PHP	(?P<name>…)	(?P=name)	不支持，只能使用\${num}，其中 num 为对应分组的数字编号
Python	(?P<name>…)	(?P=name)	\g<name>
Ruby	(?<name>…)	\k<name>	\k<name>

还有些时候，虽然想进行的操作是替换，但是不能只关心需要被替换的那段文本，而必须找到更多的文本，替换其中一部分，同时需要留下的部分。这样看起来有点多此一举，其实却是必需的，

比如删除代码文件行末的*//comment* 注释就是如此：在许多编程语言中，都可以在行末添加*//comment* 注释，它以//开头，中间可以出现任何字符，直到该行结束为止。

```
替换前
String s = "some text"; //initialization
替换后
String s = "some text";
```

要去掉末尾的注释，最容易想到的就是用`//.*$`匹配注释文本（必须设定多行模式，否则`$`无法匹配文本内部的行结束位置），将它替换为空字符串。这种做法应对上面的代码当然没问题，但遇到有些情况就会出错。

```
替换前
String url = "http://somehost.com"; //initialization
替换后
String url = "http:
```

解决办法似乎是要给正则表达式`//.*$`加上限制，在`//`匹配之后、`$`匹配之前，不能再出现/，所以把表达式改为`//[^/]*$`。这样解决了之前的问题，但是在有些情况下还是会出错。

```
替换前
String url = "http://somehost.com";
替换后
String url = "http:
```

或许可以再加点限制，在`//`匹配之后、`$`匹配之前，不但不能出现/，也不能出现"，所以把表达式改为`//[^/"]*$`，就解决了上面的问题。不幸的是，这个表达式还有问题，有些情况下，替换前后不会发生任何变化。

```
替换前
String s = "some text"; //initialization with "some text"
替换后
String s = "some text"; //initialization with "some text"
```

仔细思考就会发现，虽然注释部分是从`//`开头，到`$`结尾，但为了准确判断它，不能只着眼于注释本身，而必须考虑更多：粗略来说，//确实是注释的开头，但它也有可能出现在双引号字符串内部，或者说，在真正的程序代码中。遇到这样的//，应当忽略——或者说，应当忽略双引号字符串内部的//。

那么换个角度，从整行文本的开头来看：首先是可能出现的真正的程序代码，如果//出现在之中，肯定是出现在双引号字符串内部（暂时不考虑某些语言中单引号字符串的情况）；之后，如果出现了//，则肯定是注释部分的开始。

可以将真正程序代码的文本分为三种情况：单独出现的/、双引号字符串、/和"之外的文本，将对应的三个表达式综合起来，就得到匹配真正程序代码的表达式，再加上匹配末尾的`//.*$`，

就得到了分段匹配整行的正则表达式，整个表达式的匹配过程如表 9-7 所示。

表 9–7 匹配包含注释的代码行的正则表达式

部分		表达式
程序	单独出现的/	(?<!/)/(?!/)
	双引号字符串	"(\\.|[^\\"])*"
	/和"之外的文本	[^/"]
注释		//.*$
综合		^((?<!/)/(?!/)|"(\\.|[^\\"])*"|[^/”])*//.*$

注 1：虽然源代码中并不是每一行之后都会有注释，但在这个表达式中，匹配注释部分的`//.*`并没有用量词限定——如果这一行不包含注释，则根本不需要进行替换处理，所以正则表达式无法匹配，反而是提高了效率。

注 2：必须指定多行模式，同时用`^`和`$`来限定这个表达式只匹配一行文本。

这个表达式匹配了整行的文本，为去掉末尾的注释，必须决定保留下哪些内容。在表达式中，匹配程序语句的部分是`^((?<!/)/(?!/)|"(\\.|[^\\"])*"|[^/"])*`，在第 47 页提到，如果某个捕获分组使用了`*`量词，则对应分组保留的是最后一次匹配时捕获的文本，所以如果引用这个分组匹配的文本，得到的只是其中某个子表达式匹配的文本，解决办法是给这个表达式加上括号，这样编号为 1 的分组就保存了整个程序语句。从例 9-8 可以看到，这个表达式确实可以准确删除所有的注释。

例 9–8 准确删除注释

```
wholelineRegex = r"(?m)^(((?<!/)/(?!/)|\"(\\.|[^\\\"])*\"|[^/\"])*)//.*$"
print re.sub(wholelineRegex, r"\1", "String s = \"some text\"; //initialization\"")
String s = "some text";
print re.sub(wholelineRegex, r"\1", "String url = \"http://somehost.com\";
//initialization")
String url = "http://somehost.com";
print re.sub(wholelineRegex, r"\1", "String url = \"http://somehost.com\";")
String url = "http://somehost.com";
print re.sub(wholelineRegex, r"\1", "String s = \"some text\"; //initialization with
\"some text\"")
String s = "some text";
```

一般来说，*replacement* 字符串足够表示替换的结果了。不过有时也需要进行更复杂的替换，所以在有些语言的正则表达式替换方法中，*replacement* 可以是字符串，也可以是一个回调函数

（一般记为 *callback*），比如 Python 就是如此，下面举两个例子。

假设文本中包含一些浮点数，其格式都是 2.618、10.23 之类的，不包含符号，而且整数部分、小数点必定出现，现在希望处理所有的浮点数，只保留一位小数，而且做四舍五入处理，也就是说，2.618 要变成 3.0，而 10.23 要变为 10.2。

匹配浮点数的正则表达式很简单，粗略记为`\d+\.\d+`，但无论如何编写 *replacement* 字符串都无法进行四舍五入处理；如果使用 *replacement* 函数，则非常简单，它接收的是一个 MatchObject 对象，返回一个字符串。程序代码如例 9-9 所示。

例 9–9　使用 replacement 函数进行四舍五入处理

```
#先定义四舍五入处理的函数
def processFloat(matchObj) :
   matchedStr = matchObj.group(0)
   #小数点之后数字为 01234 的情况
   if re.search("\\.[0-4]", matchedStr) != None :
      #保留整数部分和小数第一位
      return "%.1f" % float(matchedStr)
   else :
      #其他情况，取整数部分+1，小数第一位补 0
      return "%.1f" % (int(float(matchedStr))+1)

print re.sub(r"\d+\.\d+", processFloat, "2.618 10.236")
3.0 10.2
```

还有更复杂的情况：有些网页为了防止抓取，在源代码中将文字都用&#*dec*;或者&#x*hex*;这种形式的 HTML Entity 表示，其中的 *dec* 或 *hex* 分别是十进制和十六进制编码值。用户在浏览器中看到的“收发”两个字，对应的 HTML 代码可能是收发或者收发，其中 6536 和 25910 分别是“收”的 Unicode 码值的十进制和十六进制编码值，而 53d1 和 21457 分别是“发”的 Unicode 码值的十进制和十六进制编码值。为了获取网页的内容，需要将 HTML Entity 还原为对应的文字。对于&#*dec*;或者&#x*hex*;这类文本的匹配，正则表达式很擅长，但是将某个编码值转换为对应的文字正则表达式却是无能为力的。这种时候，用 *replacement* 函数处理起来却非常简单：正则表达式是`&#(\d+|x[0-9a-fA-F]+);`，`&#`是必须出现的，然后是编码值字符串，如果是十进制的，则用`\d+`匹配；如果是十六进制的，则用`x[0-9a-fA-F]+`匹配，最后的`;`也是必须出现的。注意这是一个捕获型括号，匹配完成后通过编号为 1 的分组可以直接提取出编码值。完整代码见例 9-10。

例 9–10　使用 replacement 函数转换 HTML Entity

```
def htmlEntity2Char(matchObj) :
   if matchObj.group(0).startswith("&#x"):
      #十六进制
```

```
        return unichr(int(matchObj.group(1)[1:], 16))
        #上一行的[1:]是为了去掉开头的 x
    else :
        #十进制
        return unichr(int(matchObj.group(1)))

print re.sub(r"&#x?([0-9a-fA-F]+);", htmlEntity2Char, "&#x6536;&#x53D1;")
```

收发

```
Print.re.sub(r"&#x?([0-9a-fA-F]+);", htmlEntity2Char, "&#25910;&#21457;")
```

收发

在正则表达式替换时，将 ***replacement*** 参数设定为回调函数的做法，Python 和 JavaScript 都支持，.NET 也支持，不过.NET 通过 `deletgate`（可以理解为函数指针）实现回调函数的功能。具体的做法请参考第 10 章。

请注意，在实际开发中，应当谨慎采用替换中使用回调函数的做法，因为这样回调函数有能力执行更复杂的代码，所以可能会给恶意代码留下漏洞，带来安全性问题。

9.2.4 切分

切分也是正则表达式的常见操作之一，切分操作一般需要两个参数，即 ***regex*** 和 ***string***，它以 ***regex*** 能匹配的文本为间隔，将 ***string*** 切分开来。它返回的通常是一个数组，元素是切分之后的片段。表 9-8 中列出了常用语言中的切分方法。

表 9–8 常用语言中的切分方法

语言	方法	备注
.NET	`Regex.`**`Split`**`(`*input*`, `*pattern*`)`	静态方法
	`Regex.`**`Split`**`(`*input*`)`	`Regex` 实例事先生成
Java	`String.`**`split`**`(`*regex*`)`	静态方法
	`Pattern.`**`split`**`(`*input*`)`	`Pattern` 实例事先生成
JavaScript	`string.`**`split`**`(`*RegExp*`)`	
PHP	**`preg_split`**`(`*pattern*`, `*subject*`)`	
Python	`re.`**`split`**`(`*pattern*`, `*string*`)`	
Ruby	`string.`**`split`**`(`*Regexp*`)`	

切分操作的简单示例是切分单词，尤其是英文文本的单词。英文文本用空格分隔各个单词，比如 `This is a example of using regex`，其中包含 7 个单词，用 6 段空白分隔——其中的空白可能是空格符，也可能是制表符，还有可能是换行符。不过，无论这些空白字符是什么，都可以由正则表达式`\s+`匹配；所以以这些空白为间隔，就可以把文本分隔为 7 段，每段一个单词，如例 9-11 所示。

例 9–11　正则表达式切分

```
print re.split(r"\\s+", "This is a example of using regex")
['This', 'is', 'a', 'example', 'of', 'using', 'regex']
```

这只是最简单的情况，似乎不用正则表达式也不难做到，依据正则表达式的切分其实可以完成更复杂的任务，比如将一个中英文混排的句子按照标点符号切分成多个分句。中英文混排文本如下：

```
I have read Tao, and learned a lot from it. 我读了《道德经》，从中学到了很多。
```

希望将它切分为 4 个分句：

```
I have read Tao
and learned a lot from it
我读了《道德经》
从中学到了很多
```

如果仅仅使用字符串操作，这几乎是不可能完成的，因为既要知道识别全角、半角的逗号和句号用来切分，还要知道书名号虽然是标点符号，但不能用作切分。但使用正则表达式则非常简单，因为正则表达式提供了对 Unicode Property 的支持（因为 Python 不支持 Unicode Property，所以这里使用 Java 举例），Unicode Property 按照字符的功能分类，`\p{Po}`表示“除横线、括号、引号和连接符之外的任何标点符号”（☞ 130），代码见例 9-12。

例 9–12　借助 Unicode Property 切分

```
String str = "I have read Tao, and learned a lot from it. 我读了《道德经》，从中学到了很多。";
for (String s : str.split("\\p{Po}")) {
 System.out.println(s);
}
```

```
I have read Tao
and learned a lot from it
我读了《道德经》
从中学到了很多
```

最后简要介绍切分的次数限制。通常，切分会在所有匹配发生的地方进行，假设正则表达式能匹配 *n* 次，就切分 *n* 次，结果数组包含 *n*+1 个元素。不过，通常也可以用一个可选参数限制切分，将它设定为一个小于 *n* 的正数，则会进行 *n*-1 次切分（只有 Python 是例外，它会切分 *n* 次），返回数组的最后元素包含了“正则表达式第 *n*-1 次匹配右侧的所有文本”。

有的时候必须要用到这种功能，在例 9-13 中，SQL 日志文件的每一行都包含三个字段，分

别是发生时间、执行时间、SQL 语句，以逗号分隔，而 SQL 语句中可能就包含逗号，这时需要显式限定切分次数，否则就会麻烦很多。

例 9–13 借助 Unicode Property 切分

```
logline = "14:59:35,0.133,select name,title from employee"
#没有设定切分次数
print re.split(r",", logline)
['14:59:35', '0.133', ' select name', 'title from employee']
#显式设定切分次数
print re.split(r",", logline, 2)
['14:59:35', '0.133', ' select name,title from employee']
```

9.3 正则表达式的优化建议

许多人认为正则表达式很强大、很快，现在的电脑运行速度很快，所以效率是无所谓的，但不同的正则表达式，不同的使用方法，所消耗的时间/空间可能相差几十倍甚至上百倍，在某些环境下（比如在移动设备上），这非常重要。本节整理了一些关于正则表达式的优化建议，合理使用它们有利于提高效率。[1]

在阅读和掌握这些建议之前，应当明确两点：首先，在正则表达式的匹配中，**正确性是优先于效率的**，常见的步骤是先保证正确，再优化效率；其次，如果效率不是严重的问题，就应当**仔细审视优化的价值**，因为有些优化技巧是需要改动正则表达式，从而影响可读性的，这么做是否值得，取决于具体的情况。

9.3.1 使用缓存

第 8 章已经介绍过正则表达式的匹配原理，生成正则表达式对象（也就是从字符串形式的表达式生成自动机）是非常复杂的过程。所以如果一个表达式需要反复使用，最好的办法就是将生成的表达式对象缓存起来。

在函数式处理中，正则表达式的缓存一般由语言自动完成，比如 PHP 就可以缓存 4000 多个表达式。不过，这种机制对用户是透明的，一般用户完全不必关心。

在面向对象式处理中，正则表达式对象的生成可以由用户手工控制，例如 Python 中的 `re.compile(`*`pattern`*`)`、Java 中的 `Pattern.compile(`*`regex`*`)`、.NET 中的 `new Regex(`*`regex`*`)`。如果正则表达式对象需要反复用到，就应当用变量将其保存起来，每次直接使用，避免重复编译。

[1] 为了节省篇幅，这里没有给出真正程序的测试数据，有兴趣的读者可以参考《精通正则表达式》（第 3 版）第 6 章的“性能测试”。

比如Java中的正则表达式替换，最简单的办法是String.replaceAll(*regex*, *replacement*)，这是String类的静态方法，它的参数*regex*和*replacement*都是String对象。但是另一方面，也可以调用Matcher.replaceAll(*replacement*)方法，Matcher对象由编译好的Pattern对象和String对象生成。如果要对多个String进行替换，先编译好Pattern对象，再对每个String生成Matcher，调用成员方法Matcher.replaceAll(replacement)，效率就高很多。

而.NET中还提供了更精确的缓存控制，可以人工设定Regex.CacheSize的值，指定缓存最近的*n*个正则表达式，根据场景选择合适的值，有可能大大提高效率。

9.3.2　尽量准确地表达意图

正则表达式中提供了许多结构（字符组、多选结构、量词、环视结构等），许多结构的功能存在重叠，有些情况下，一些结构的功能可以完全由其他结构提供。这时候，不应该图省事使用“功能更全面”的结构，而应该恰当选取最合适（也就是更“专用”）的结构。

最常见的例子是多选结构和字符组，多选结构的功能完全覆盖了字符组。`(a|b|c|d)`和`[abcd]`能匹配的文本是完全一样的，但是，多选结构是用于处理“可能出现若干个表达式”的，而字符组是专用于处理“可能出现若干个字符”的。针对例子中的这种情况，字符组的效率会远远高于多选结构，所以应当使用`[abcd]`而不是`(a|b|c|d)`。

当然，字符组也不是任何时候都推荐使用。有些人不明白转义的规则，但知道字符组内部除去-字符，所有元字符都失去了特殊含义，就用字符组来“转义”，比如要匹配点号`.`，不使用`\.`，而是用`[.]`，这种做法同样会影响效率。

在第3章讲过，一旦使用括号，无论是用于分组，还是用于多选结构，都会产生副作用——匹配时会把文本保存起来，匹配完成之后通过对应的编号访问。但是许多时候，括号仅仅用来分组，匹配完成之后并不关心括号内的子表达式匹配的文本。这时候，就应当把捕获型括号`(`*regex*`)`改为非捕获型括号`(?:`*regex*`)`。不过，捕获型括号`(`*regex*`)`更美观，也更符合习惯，所以我推荐的做法是，在编写正则表达式时，先使用捕获型括号，确保准确无误之后，再改为非捕获型括号。

另外，如果要匹配任意的英文大小写字母，既可以使用`[a-zA-Z]`，也可以使用`[a-z]`，然后指定不区分大小写模式。相对来说，前者更明确，所以效率也更高些（当然这个例子最合适的是匹配“任意”字母的情况，如果要匹配的是确定单词，比如cat，明确写出就是`[cC][aA][tT]`，明显不如`cat`直观，更不用说tomorrow之类的单词了）。但是，只有在要处理的文本特别多的情况下，这种优化才会带来比较大的效率提升，所以这个技巧应当斟酌使用，根据具体情况进行选择。

9.3.3　避免重复匹配

正则表达式非常灵活，同一个表达式往往可以匹配很多种不同形态的文本。但是，如果一个

表达式中包含若干个“能匹配不同形态文本”的子表达式，而这些子表达式能匹配的文本又有重叠，就可能造成效率问题。

最简单的例子是表达式 `a+a+`。上一章讲解过正则表达式的匹配原理：因为 `a+`中`+`的存在，匹配时每遇到一个字符 a 都需要保存备用状态，以应付可能的回溯。这样，对于同一个字符，两个 `a+`保存的备用状态数量就比单个 `a+`要多出很多，而这其实是不必要的（某段文本如果不能被第一个 `a+`匹配，必定也不能被第二个 `a+`匹配），所以应当避免出现这样的表达式。

这类问题看起来很简单，在正常情况下很难有人写 `a+a+`之类的表达式，甚至 `a+a*`也不会出现，但是`[0-9]+\w+`之类的表达式则比较容易遇到，其中`[0-9]`和`\w` 存在重复匹配的问题。更复杂的情况是类似`(…[0-9]+)(\w+…)`之类的表达式，看起来`(…[0-9]+)`和`(\w+…)`的逻辑都足够清楚，各部分前后连接的地方却出现了重复匹配的问题。在构造复杂表达式的过程中，很容易出现这类问题，一定要多加注意。

更复杂的重复匹配的问题来自多选结构。之前提到过，在多选结构中一定要尽力避免各个多选分支可能匹配相同的文本。在之前提到的匹配双引号字符串的例子中，使用多选结构`(\\.|[^\\"])`匹配起始双引号和结束双引号之间的文本，其中第一个多选分支`\\.`匹配任何转义序列，第二个多选分支本意是匹配其他字符（也就是除去"之外的其他字符），不仔细思考可能就会使用字符组`[^"]`，这样不但会降低效率，还会造成匹配错误。

重复匹配的情况看似麻烦，其实并非如此。总结起来，主要有两条规则，如果能做到，则大部分重复匹配的问题都可以避免。

（1）凡类似 *`regex1*regex2*`*（量词不限于`*`）的表达式，都要尽量避免 *`regex1`* 和 *`regex2`* 能匹配相同的内容。

（2）凡类似`(regex1|regex2)`的表达式，都要尽力避免 *`regex1`* 和 *`regex2`* 能匹配相同的内容。

9.3.4 独立出文本和锚点

正则表达式包含多种复杂的结构，提供了匹配文本的各种可能性。可能性增加的代价就是降低了处理的速度。根据之前讲解的知识，正则表达式匹配都要经过“编译正则表达式”和“逐次试错”的过程，普通的字符串操作没有这么复杂，按照目前普遍采用的Boyer-Moore字符串检索算法[1]，能很快找到某个字符串在文本中的位置。

对正则表达式来说，我们常常可以发现多种“可能性”的共同点，也就是各种可能中都具有的、不变的字符串，将它们提取出来，就可以依靠普通的字符串操作，迅速定位到某个位置，再“逐次试错”。

[1] 请参考 http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm。

仔细观察正则表达式`(this|that|these|those)`可以发现，多选结构的所有多选分支都以`th`开头，所以这个表达式如果能够匹配，前两个字符必然是**th**。但正则表达式处理程序未必能“意识”到这一点，所以不妨人工将`th`提取出来，将表达式改为`th(is|at|ese|ose)`。同样，如果多选分支的结尾是相同的，依然可以人工提取出来，把表达式`(option|action)`改为`(op|ac)tion`，也有这种效果。[1]

将文本独立出来的技巧也适用于量词。比如`=+`要求等号**=**至少必须出现一次，它等价于`==*`，但是后者独立出了一个固定的字符，所以效率更高；再比如`={4}`，它等价于`====`，但后者独立出 4 个固定的字符，因此效率更高。测试数据显示，在 Python 和 Java 中，后者的效率是前者的 100 倍以上；不过，Ruby 和.NET 能自动识别两者，自动进行优化，所以两者效率是一样的。

将文本独立出来可以提高效率，同理，将锚点独立出来也可以提高匹配的效率。比如`(^this|^that)`，正则表达式处理程序未必知道两个分支都必须在字符串的起始位置（如果指定了多行模式，则是行的起始位置）开始匹配。如果将`^`独立出来，改为`^(this|that)`，也可以提高效率。

9.4　别过分依赖正则表达式

正则表达式的功能很强大，有些人学会了之后随时都想使用它。但是，过分依赖正则表达式并不是好习惯。下面列举几个过分依赖正则表达式的常见错误。

9.4.1　彻底放弃字符串操作

最常见的错误是，某个任务明明只需要字符串操作就可以完成，但非要使用正则表达式。上一章已经分析过正则表达式的匹配原理，正则表达式匹配需要先构建有穷自动机，这种操作的成本高昂。如果只需要查找；或者，如果只需要判断字符串开头以某个固定子串开头，以某个固定子串结尾，一般语言中都提供了 *startswith*()、*endswith*()之类的方法，相比之下，正则表达式即便使用了锚点，效率也要低得多。表 9-9 比较了这两类操作。

表 9–9　这些情况下，字符串操作比正则表达式更高效

字符串操作	正则表达式操作
`string.index("cat") != -1`	`re.search("cat", string) != None`
`string.startswith("cat")`	`re.search("^cat", string) != None`
`string.endswith("cat")`	`re.search("cat$", string) != None`

如果检查多个固定形式的字符串，正则表达式的效率仍然比不上简单的字符串操作。比如要

[1] 这样的代价是增加了阅读代码的难度。所以我的建议是：这类优化措施，首先必须保证正确；其次，适当的时候应当添加注释。

检查一个字符串中是否包含 one、two、three 这三个单词中的任意一个，正则表达式的解法是查找`(one|two|three)`，但它的效率比不上依次判断是否存在 one、two、three。因为简单字符串查找已经有很多高效的算法，“大而全”的正则表达式却必须在文本的每个位置尝试匹配。要查找的字符串越多，正则表达式的效率就越低。在实际应用中，经常可能要检查文本中是否包含某些不期望的文本（“敏感”词），如果文本很长，不期望出现的词很多，使用正则表达式就可能严重影响效率。

9.4.2 思维定式

另一个常见的错误是形成了思维定式，解决问题的思路被正则表达式的 API 限制。针对正则表达式的常见操作（提取、验证等），编程语言和工具都提供了对应的 API，但这并不是说只能硬性使用这些 API。

回过头看之前讲解的密码字符串匹配的例子，符合要求的密码必须满足以下 5 点要求：

（1）密码的长度在 6~12 个字符之间；

（2）只能由小写字母、阿拉伯数字、横线组成；

（3）开头和结尾不能是横线；

（4）不能全部是数字；

（5）不允许有连续（2 个及以上）的横线。

单独针对其中每个条件判断，表达式都很容易写（甚至更容易，比如“不允许有连续（2 个及以上）的横线”，可以先验证表达式中是否存在连续的横线，再对结果取“非”）。总的验证如果要成立，则必须同时满足这 5 个条件，也就是对 5 个单独验证的结果进行“与”运算。这种思路的代码见例 9-14，比起费力把 5 个条件整合到一个表达式中，这种办法要清晰多了。

例 9–14 用一组正则表达式验证

```
cond1Regex = r"\A.{6,12}\Z"
cond2Regex = r"\A.[0-9A-Za-z-]*\Z"
cond3Regex = r"(\A-|-\Z)"  #开头或结尾是横线
cond4Regex = r"\D" #出现一个非数字字符
cond5Regex = r"--" #连续的横线
#Python 中语句末尾没有;，行末的\表示同一条语句未结束
re.search(cond1Regex,  password) != None \
and re.search(cond2Regex,  password) != None \
and re.search(cond3Regex,  password) == None \ #注意，此处取非
and re.search(cond4Regex,  password) != None \
and re.search(cond5Regex,  password) == None \ #注意，此处取非
```

另一个例子是去掉行首尾的空白字符。第 67 页已经说明，最简单的办法是分两步处理，在多行模式下，分别将`^\s+`和`\s+$`能匹配的文本替换为空字符串，这种做法直观，而且效率最高；

如果非要用一个表达式替换来完成，不但可能出错，效率也无法保障，甚至可能降低很多。

从这点推广开来，正则表达式并没有规定，执行某类操作就必须使用对应的 API，同一个任务有多种完成方法。比如从日期字符串 2010-10-22 提取出年、月、日信息，既可以用表达式`(\d{4})-(\d{2})-(\d{2})`匹配再提取对应捕获分组，也可以用`-`切分，两者的结果是一样的。

9.4.3　正则表达式可以匹配各种文本

最后一个常见错误是，以为正则表达式能完成所有的任务。正则表达式的功能虽然强大，但只限于描述文本特征，而且甚至无法覆盖所有的文本特征。

如果要匹配这样的文本：`01` 或 `0011` 或 `000111`……总之是由 `0` 和 `1` 构成的字符串，连续的 `0` 在前，连续的 `1` 在后，而且 `0` 的数目和 `1` 的数目应当相等。在正则表达式中，对于这类情况应当使用量词，所以表达式应当类似`0+1+`。遗憾的是，在正则表达式中我们无法表示“限定 `0` 的量词等于限定 `1` 的量词”，所以仅仅依靠正则表达式是无法解决这类问题的。而例 9-15 的逻辑则要简单很多：先用正则表达式`(0+)(1+)`验证，通过之后，再对比这个表达式中编号为 1、2 的两个捕获分组匹配文本的长度，如果相等，则留下，否则忽略。

例 9-15　正则表达式配合其他运算进行验证

```
def validate(str) :
    matchObj = re.search(r"\A(0+)(1+)\Z", str)
    return (matchObj != None) and (len(matchObj.group(1)) == len(matchObj.group(2)))

validate("001")    # => False
validate("0011")   # => True
```

还有些时候，要匹配的文本不只具有文本特征，还有其他方面的意义要求，这时候正则表达式更是无能为力了，闰年的匹配就是如此。

一般来说，闰年就是可以被 4 整除的年份。闰年的两位尾数其实很好判断：如果十位是 0、2、4、6、8，则个位可以是 0、4、8；如果十位是 1、3、5、7、9，则个位可以是 2、6，看来，匹配闰年的表达式是`\d{2}([02468][048]|[13579][26])`（这里暂时不考虑两端的断言）。

但是，闰年还有另一条规定：尾数为 00 的年份，除非同时是整个数字能被 400 整除，否则不是闰年。所以 2000 年是闰年，而 1900 年、1800 年都不是闰年。如果要提取某段文本中所有闰年的年份，单纯使用正则表达式非常吃力。但如果先提取出所有“疑似闰年”的年份，再用一个函数判断这个年份如果尾数是 00 能否整除 400，对之前提取的年份进行过滤，则要简单许多，代码见例 9-16。

例 9–16 正则表达式配合其他运算验证闰年

```
def validateLeapYear(str) :
    return (re.search(r"\A\d{2}([02468][048]|[13579][26])\Z", str) != None) and
int(str) % 400 != 0

validateLeapYear("1986")    # => False
validateLeapYear("1988")    # => True
validateLeapYear("2000")    # => False
validateLeapYear("2100")    # => True
```

9.4.4 滥用正则表达式

正则表达式很强大，所以很多人一旦学会了正则表达式，就像掌握了万能钥匙，凡是涉及文本的处理都要“玩上”两把正则表达式才过瘾。甚至，因为正则表达式诞生比较早，在外人看来语法古怪而复杂，所以掌握了它往往会得到很多崇拜和赞誉，潜意识里也希望能多露两手。但是我们需要知道，除非你在杂技团，否则所有的技术都是用来解决问题的。有许多与文本相关的问题，用正则表达式解决并不是最佳答案。

比如最简单的验证 E-mail 地址，网上可以搜索到很多“匹配 E-mail”的正则表达式，许多都声称自己没问题。简单用用也确实没有问题，然而一旦它们成了基础服务，日后的麻烦就接踵而来。我见过很多这样的例子：没有考虑到用户名可以只有一个字母，比如 `i@somehost.com`（如果有自己的域名，开设这样的邮箱相当容易）；没有考虑到用户名中可以包含`+`或者`.`，比如 `tom.work@gmail.com`……

如果仔细读过 RFC 文档就会知道，E-mail 地址是否合法，并不是完全由文本特征来判断的，或者至少不是用简单的文本特征来判断的。也就是说，随便写一个正则表达式就能严格验证 E-mail 地址，这基本是不可能的。

怎么办？其实已经有很多现成的解法，安全可信，只是和正则表达式无关。比如在 C#中可以像例 9-17 中这样解决。

例 9–17 在 C#中准确验证 E–mail 地址

```
bool IsValidEmail(string email)
{
    try {
        var addr = new System.Net.Mail.MailAddress(email);
        return addr.Address == email;
    }
    catch {
        return false;
    }
}
```

在 Java 语言中没有这样的办法，但是 Apache Commons 中提供了各种 Validator，其中就包括 EmailValidator，所以可以像例 9-18 这样处理。

例 9–18　在 Java 中准确验证 E–mail 地址

```
import org.apache.commons.validator.EmailValidator;

bool IsValidEmail(string email)
{
    return EmailValidator.getInstance().isValid(email);
}
```

再比如，对中国大陆手机号的验证。因为手机号的格式并不统一，有时以 0 开始，有时不以 0 开始，还有些时候前面有+86 或者 86，所以很多人会想当然地用正则表达式来验证。没错，正则表达式很适合处理这样的情况。但是我们也知道，随着手机用户量不断增加，号段也在不断增加，15x、17x、18x 号段早已经不新鲜，最近 19x 号段也开始投入使用。如果仍然只想“用一个正则表达式准确验证手机号”，这个表达式就需要不断更新。更要命的是，这个表达式很难被看懂。惯常思维都是把号段视为一组三位数字的组合，比如 130、131、132…150…170…199。如果直接把它们放入多选结构，那么正则表达式会很长，如果进行合并，正则表达式又很难理解，更难准确更新。那么，有什么好办法吗？

我们都知道，软件设计中有一条原则是“关注点分离”（SOC）。在设计系统时，许多人都会遵循这条原则，但在解决具体问题时往往忘记了它。具体到这个例子，我们可以仔细看看，如果应用关注点分离原则，应该如何解决。

使用正则表达式时，关注点是应对+86 或 86 开头，手机号前面可能出现 0 的情况。除此之外，如果只是验证号段，任何一个程序员都会用简单的字符串处理来完成，稍微动动脑筋就可以知道，可以把合法号段作为配置独立出来，剩下的只是把字符串中对应号段的那几位与预先配置好的合法号段比较，如果有命中则号段验证通过，否则号段验证不通过。

所以完全可以把验证逻辑拆分成两部分：正则表达式负责验证手机号的各种形式，并负责提取号段对应位置的子字符串，交给号段验证程序验证。这样一举两得，既保证了灵活性，又降低了维护的复杂性，唯一放弃的是“秀正则表达式的机会”。不过，这是值得的。

从某些方面来看，正则表达式和设计模式差不多——要掌握它，不但要知道它能做什么，什么时候该用它，更要知道它不能做什么，什么时候不该用它。否则，即便学会了正则表达式的基本用法，你的生活也不会更幸福。

第三部分

第 10 章　.NET

在微软的.NET Framework里的C#、C++、VB.NET等各种语言中，正则表达式的包是统一的，都是`System.Text.RegularExpressions`；而且，基本的对象和方法在各语言中差异不大，所以在某种语言中编写的关于正则表达式的程序，可以很方便地移植到.NET的其他语言中。为节省篇幅，本章的大多数示例使用C#，VB.NET的代码与之没有本质差别，所以不再重复。[1]

10.1　预备知识

.NET 语言中的正则表达式相关类都存在于 `System.Text.RegularExpressions` 中，如果要使用正则表达式，必须首先导入它，最简单的办法是在程序文件头部这样写：

C#
```
using System.Text.RegularExpressions;
```
VB.NET
```
Imports System.Text.RegularExpressions;
```

对应正则表达式的类名称为`Regex`，这个类会在下面详细介绍，这里先介绍.NET中的`Regex.IsMatch(`*input*`, `*regex*`)`方法[2]，它返回一个布尔值，表示*regex*对应的表达式能否在*input*中找到匹配。即使只能匹配一个子串，也会返回True，如果要检查整个字符串能否由表达式匹配，可以在表达式首尾加上`\A`和`\z`。

```
Regex.IsMatch("1", "\\d");          // => True
Regex.IsMatch("1a", "\\d");         // => True
Regex.IsMatch("1", "\\A\\d\\z");    // => True
Regex.IsMatch("1a", "\\A\\d\\z");   // => False
```

请注意其中的字符组简记法和锚点都使用了两个反斜线字符，因为需要转义。其实 C#的 String 也提供了**原生字符串**（Verbatim String），它不需要考虑字符串转义，所以非常适合表示正则表达式：正则表达式是怎样的，原生字符串就写成什么样。具体做法是在 String 开头的双引号

[1] 详细文档可参考 http://msdn.microsoft.com/en-us/library/hs600312%28v=vs.71%29.aspx。

[2] 在.NET 中的正则 API 的参数中，字符串在前，正则表达式在后，顺序不同于 Python、Java、PHP 等语言，如果要同时在多种语言中编程，请留意这一点。

之前添加@，为方便阅读，本章的示例代码都采用这种写法。

```
Regex.IsMatch("1", @"\d");          // => True
Regex.IsMatch("1a", @"\d");         // => True
Regex.IsMatch("1", @"\A\d\z");      // => True
Regex.IsMatch("1a", @"\A\d\z");     // => False
```

因为原生字符串不识别任何反斜线转义序列，字符串中使用的\n、\t 表示法都无法用在原生字符串中，所以本章只对正则表达式参数使用原生字符串，而不对字符串参数使用原生字符串。

VB.NET 中的字符串比较特别，处理字符串文字时不识别任何转义序列，所以可以理解为，VB.NET 中的所有字符串天生就是原生字符串，唯一例外的文字是紧密相连的两个双引号""，它表示一个双引号"。下面给出了几个 VB.NET 中的例子。

```
Regex.IsMatch("1", "\d");       // => True
Regex.IsMatch("""","""");       // => True
```

10.2 正则功能详解

10.2.1 列表

表 10-1 中列出了.NET 中的正则功能。

表 10–1 .NET 中的正则功能列表

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	完全支持 Unicode
POSIX 字符组 ☞15	`[:digit:]`	不支持
Unicode 属性 ☞127	`\p{…}`	完全支持 Unicode
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	采用 Unicode 规则
行起始位置 ☞62	`^` `\A`	支持
行结束位置 ☞62	`$` `\z` `\Z`	支持
单词边界 ☞123	`\b` `\B`	采用 Unicode 规则
顺序环视 ☞69	`(?=…)` `(?!…)`	支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	支持
匹配模式 ☞83	`i m s x`	支持
模式作用范围 ☞91	`(?modifier)` `(?-modifier)`	支持
纯文本模式 ☞101	`\Q…\E`	不支持
捕获分组及引用☞44	`(…)` `\num` `$num`	支持
命名分组 ☞53	`(?<name>…)` `\k{name}` `${name}`	支持
非捕获分组 ☞55	`(?:…)`	支持

（续表）

功能	记法	说明
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

10.2.2　字符组

在.NET 语言中，**所有的字符都是 Unicode 字符**，所以在字符组中可以直接使用中文，因而不会出现多字节字符错误匹配的问题；另一方面，因为所有字符都是 Unicode 字符，所以点号也可以匹配中文字符。

```
Regex.IsMatch(".",@"\A 你\z");        // => True
Regex.IsMatch("遭", @"\A[正则]\z");   // => False
```

值得注意的是，.NET 语言中的字符组可以进行集合运算，此功能非常适合进行字符组的“减”运算——如果需要匹配英文中所有小写的辅音字母，也就是从 26 个小写字母中“减去”5 个元音字母，可以写作`[a-z-[aeiou]]`（注意不是`[[a-z]-[aeiou]]`）。它的意思是，“从 a 到 z 的所有字符（也就是小写英文字符）”与“除 aeiou 之外的任何字符”取交集，实际结果就是所有的小写辅音字母。

```
Regex.IsMatch("a", @"[a-z-[aeiou]]");     // => False
Regex.IsMatch("b", @"[a-z-[aeiou]]");     // => True
```

也可以用这个字符组来匹配“英文大小写字母和数字（不包括下画线）”。

```
Regex.IsMatch("a", @"[\w-[_]]");     // => True
Regex.IsMatch("0", @"[\w-[_]]");     // => True
Regex.IsMatch("_", @"[\w-[_]]");     // => False
```

.NET 中可以用\u*hex* 指定 Unicode 码值，其中 *hex* 是 4 位十六进制数。所以，在 Unicode 编码环境下可以用字符组`[\u4E00-\u9FFF]`匹配所有中文字符。

```
Regex.IsMatch("我", @"[\u4E00-\u9FFF]"); // => True
```

10.2.3　Unicode 属性

.NET 对 Unicode 字符组的支持比较好，它支持 Unicode Property。这样，用`\p{N}`就可以匹配所有的数字字符，包括中文的全角数字字符，比如 0 、 1 、 2 …；`\p{P}`也可以匹配各种标点符号，包括中文的冒号：、书名号《和》等。

```
Regex.IsMatch("１", @"\p{N}");    // => True
Regex.IsMatch("：", @"\p{P}");    // => True
Regex.IsMatch("《", @"\p{P}");    // => True
```

.NET也支持Unicode Block，其记法是以Is为前缀的，这与Java不一样，使用时应注意[1]，比如匹配中文字符的Unicode Block，就应当写作`IsCJKUnifiedIdeographs`。

```
Regex.IsMatch("我", @"\p{IsCJKUnifiedIdeographs}");    // => True
```

关于 Unicode 属性的细节，请参考第 130 页。

10.2.4 字符组简记法

在.NET 的正则表达式中，`\d`、`\D`、`\w`、`\W`、`\s`、`\S`都使用 Unicode 匹配规则。也就是说，`\d`不仅仅等价于`[0-9]`，还可以匹配其他语言中的数字字符；`\w`不仅仅等价于`[0-9a-zA-Z_]`，还可以匹配其他语言中的文字字符；`\s`也可以匹配 ASCII 编码之外的空白字符。

```
//半角字符
Regex.IsMatch("1", @"\d");      // => True
Regex.IsMatch("a", @"\w");      // => True
Regex.IsMatch(" ", @"\s");      // => True

//全角字符及中文字符
Regex.IsMatch("１", @"\d");     // => True
Regex.IsMatch("我", @"\w");     // => True
Regex.IsMatch("　", @"\s");     // => True
```

如果要“恢复”这几个字符组简记法的 ASCII 匹配规则，可以指定使用 `ECMAScript` 模式，因为它没有对应的模式修饰符，所以只能以预定义常量 `RegexOptions.ECMAScript` 的方式指定。

```
//全角字符及中文字符
Regex.IsMatch("１", @"\d", RegexOptions.ECMAScript); // => False
Regex.IsMatch("我", @"\w", RegexOptions.ECMAScript); // => False
Regex.IsMatch("　", @"\s", RegexOptions.ECMAScript); // => False
```

10.2.5 单词边界

.NET 正则表达式中的单词边界采用 Unicode 匹配规则，也就是说，如果`\b`能匹配，则一侧必须出现`\w`能匹配的字符，另一侧不能出现`\w`能匹配的字符（不是“另一侧出现`\w`不能匹配的字符”）。

```
//英文字符-半角标点
Regex.IsMatch("a,", @"a\b");     // => True
//英文字符-全角标点
Regex.IsMatch("a ，", @"a\b");   // => True
//英文字符-结束
```

[1] Unicode Block 的细节可参考 http://msdn.microsoft.com/en-us/library/20bw873z.aspx#SupportedNamedBlocks。

```
Regex.IsMatch("a", @"a\b");       // => True
//中文字符-半角标点
Regex.IsMatch("我,", @"我\b");   // => True
//中文字符-全角标点
Regex.IsMatch("我，", @"我\b");  // => True
//中文字符-空字符串
Regex.IsMatch("我\b ", @"我");   // => True
//全角标点-结束（未出现单词字符）
Regex.IsMatch("，", @"，\b");    // => False
//中文字符-英文字符（未出现单词字符）
Regex.IsMatch("我 a", @"我\b");  // => False
```

指定 ECMAScript 模式会影响\w的匹配，所以\b的匹配也同样会受到影响。

```
//英文字符-半角标点
Regex.IsMatch("a,", @"a\b", RegexOptions.ECMAScript);      // => True
//英文字符-全角标点
Regex.IsMatch("a ，", @"a\b", RegexOptions.ECMAScript);    // => True
//英文字符-结束
Regex.IsMatch("a", @"a\b", RegexOptions.ECMAScript);       // => True
//中文字符-半角标点
Regex.IsMatch("我,", @"我\b", RegexOptions.ECMAScript);    // => False
//中文字符-全角标点
Regex.IsMatch("我，", @"我\b", RegexOptions.ECMAScript);   // => False
//中文字符-空字符串
Regex.IsMatch("我\b ", @"我", RegexOptions.ECMAScript);    // => False
//全角标点-结束（未出现单词字符）
Regex.IsMatch("，", @"，\b", RegexOptions.ECMAScript);     // => False
//中文字符-英文字符（未出现单词字符）
Regex.IsMatch("我 a", @"我\b", RegexOptions.ECMAScript);   // => True
```

10.2.6　行起始/结束位置

^匹配的是“行开始的位置”，默认情况下它只能匹配“整个字符串的开始位置”，如果指定使用多行模式（Multiline Mode），^可以匹配字符串内部的文本行的开始位置；而\A无论在什么情况下，都匹配“整个字符串的开始位置”。

```
Regex.IsMatch("1\n2\n", @"^1");       // => True
Regex.IsMatch("1\n2\n", @"^2");       // => False
Regex.IsMatch("1\n2\n", @"(?m)^2");   // => True

Regex.IsMatch("1\n2\n", @"\A2");      // => True
Regex.IsMatch("1\n2\n", @"\A2");      // => False
Regex.IsMatch("1\n2\n", @"(?m)\A2");  // => False
```

在默认情况下，.NET 中的$、\Z、\z匹配的是整个字符串的结束位置（如果结束位置有换

行符，则$和\Z匹配这个换行符之前的位置），多行模式只会影响到$的匹配，在这种模式下它可以匹配文本内部行的结束位置。所以如果要做严格准确的验证，应当使用\z。

```
Regex.IsMatch("1\n", @"1$");          // => True
Regex.IsMatch("1\n2", @"1$");         // => False
Regex.IsMatch("1\n2", @"(?m)1$");     // => True

Regex.IsMatch("1\n", @"1\Z");         // => True
Regex.IsMatch("1\n2", @"1\Z");        // => False
Regex.IsMatch("1\n2", @"(?m)1\Z");    // => False

Regex.IsMatch("1", @"1\z");           // => True
Regex.IsMatch("1\n", @"1\z");         // => False
Regex.IsMatch("1\n", @"(?m)1\z");     // => False
```

10.2.7 环视

.NET 中的肯定顺序环视(?=…)和否定顺序环视(?!…)内可以使用任意形式的子表达式。

```
Regex.IsMatch("cab", @"c(?=ab+)");         // => True
Regex.IsMatch("ccd", @"c(?=(ab+|cd))");    // => True
Regex.IsMatch("cabbbb", @"c(?=(ab+|cd))"); // => True
```

.NET 的逆序环视特别值得表扬——在目前所有的编程语言中，只有.NET 对逆序环视没有任何限制，也就是说，可以在.NET 的逆序环视中使用任何形式的表达式。

```
Regex.IsMatch("ab", @"ab(?<=ab)");              // => True
Regex.IsMatch("cd", @"cd+(?<=((ab)*cd|cd))");   // => True
Regex.IsMatch("cd", @"cd(?<=c+d?)");            // => True
```

虽然.NET 对逆序环视的支持相当完备，但我并不推荐在逆序环视中使用特别复杂的表达式，因为这样可能会严重影响性能。

10.2.8 匹配模式

表 10-2 中列出了.NET 中的匹配模式。

表 10–2 .NET 中的匹配模式

常量	修饰符	说明
IgnoreCase	i	不区分 ASCII 字符的大小写
IgnorePatternWhitespace	x	允许正则表达式中出现注释，表达式中的空白字符，以及#开始到行末的文本，都视为注释

（续表）

常量	修饰符	说明
Multiline	m	允许^和$不仅仅匹配字符串的起始和结束位置，还可以匹配字符串内部文本行的起始和结束位置
Singleline	s	允许点号.匹配任何字符，包括换行符
ExplicitCapture	n	只为命名分组保留匹配的文本，非命名分组不保留匹配的文本
ECMAScript	无	指定字符组简记法\d \D \w \W \s \S 采用 ASCII 匹配规则
Compiled	无	将正则表达式直接编译为底层 MSIL（Microsoft Intermediate Language），会降低启动速度，增加内存消耗，但匹配速度能大幅提升，适合将特别常用的正则表达式编译到 DLL 中重用

匹配模式可以在表达式中以(?*modifier*)指定，也可以在生成 Regex 对象时，将预定义常量作为参数指定。

```
Regex.IsMatch("A", @"a");          // => False
Regex.IsMatch("A", @"(?i)a");      // => True
Regex.IsMatch("A", @"a", RegexOptions.IgnoreCase); // => True
```

.NET 也支持用(?-*modifier*)停用某个模式。

```
Regex.IsMatch("aBb", @"a(?i)b(?-i)b"); // => True
Regex.IsMatch("aBB", @"a(?i)b(?-i)b"); // => False
```

10.2.9　捕获分组的引用

在.NET 中，如果要在正则表达式内部引用捕获分组，应当使用*num* 记法，其中 *num* 为对应捕获分组的编号。

```
Regex.IsMatch("ab", @"([a-z])\1"); // => False
Regex.IsMatch("aa", @"([a-z])\1"); // => True
```

当然也可以使用命名分组，这样看起来更清楚，在表达式中引用命名分组的记法是\k<*name*>。

```
Regex.IsMatch("ab", @"(?<char>[a-z])\k<char>"); // => False
Regex.IsMatch("aa", @"(?<char>[a-z])\k<char>"); // => True
```

如果要在替换时引用捕获分组，应当使用$*num* 记法。

```
Regex.Replace("2010-12-20", @"(\d{4})-(\d{2})-(\d{2})", "$2/$3/$1");
12/20/2010
```

当然也可以使用命名分组，但是在 *replacement* 字符串中引用的记法不同于在表达式中，应当写为${*name*}。

```
Regex.Replace ("2010-12-20", @"(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})", @"${month}/
${day}/${year}");
12/20/2010
```

如果要在 *replacement* 字符串中使用$字符，必须将它转义为$$。

```
Regex.Replace("the price is 12.99", @"\d+\.\d{0,2}", "$$$0");
the price is $12.99
```

10.3 正则 API 简介

.NET 语言中的正则表达式相关类都存在于 System.Text.RegularExpressions 中，通常用到的是这几个类：Regex、Match、Group，关系如图 10-1 所示。Regex 对应正则表达式，一个 Regex 对象与一个 String 对象关联，生成一个 Match 对象，它表示 Regex 在 String 中的一次匹配，对它调用 *NextMatch*()方法，可以获得下一次匹配的 Match 对象；借助 Match 获得对应的 Group 对象，可以访问当次匹配时的分组捕获信息。

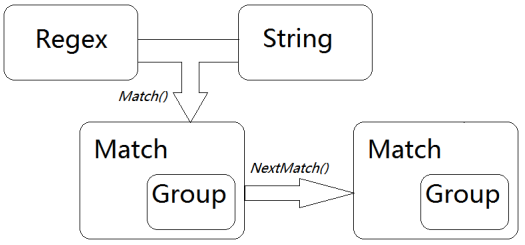


图 10–1 .NET 中与正则表达式相关的类

```
Regex regex = new Regex(@"\d{4}-\d{2}-\d{2}");
Match match = regex.Match("2010-12-20 2011-02-14");
int[] groupNums = regex.GetGroupNumbers();
while (match.Success) {
   Console.WriteLine(match.Groups[0].Value);
   match = match.NextMatch();
}
2010-12-20
2011-02-14
```

10.3.1 Regex

Regex 是.NET 中的“正则表达式对象”（之前介绍过，虽然正则表达式是用字符串形式表示的，但它并不等于字符串）。要使用正则表达式，首先必须从字符串“编译”（也可以说“生成”）

出 Regex 对象，常用方法是这样的：

```
Regex regex = new Regex(@"ab+");
```

如果要指定匹配模式，可以在表达式中使用修饰符(?*modifier*)指定，也可以使用预定义常量。下面的两个 Regex 对象虽然生成方式不同，却是等价的。

```
Regex regex = new Regex (@"(?i)ab+");
Regex regex = new Regex (@"ab+", Pattern.CASE_INSENSITIVE);
```

如果要同时指定多种模式，可以连写模式修饰符，也可以直接用|将预定义常量连接起来。下面的两个 Regex 对象的功能是完全相同的。

```
Regex regex = new Regex (@"(?is)ab+");
Regex regex = new Regex (@"ab+", RegexOptions.IgnoreCase | RegexOptions.Singleline);
```

下面介绍 Regex 类的主要成员方法。

10.3.1.1　Static String Escape(String text)

这个方法用来取消字符串 *text* 中所有转义字符的特殊含义。如果需要取消某个字符串中所有元字符的特殊含义（比如从外界读入的字符串，用于正则表达式搜索），就可以使用这个方法。

```
Regex.IsMatch("aacb", @"a*.b");                 // => True
Regex.IsMatch("aacb", Regex.Escape(@"a*.b"));   // => False

Regex.IsMatch("a*.b", @"a*.b");                 // => False
Regex.IsMatch("a*.b", Regex.Escape(@"a*.b"));   // => False
```

10.3.1.2　String[] GetGroupNames()

这个方法用来返回当前 Regex 对象中各命名分组的名字（对应整个表达式的默认分组名字是 0），如果只有捕获分组没有命名分组，则将捕获分组的数字编号 1、2、3 之类作为字符串返回。

```
Regex regex = new Regex(@"(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})");
foreach (String name in regex.GetGroupNames()) {
 Console.Write(name + " ");
}
0 year month day

Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
foreach (String name in regex.GetGroupNames()) {
 Console.WriteLine(name + " ");
}
0 1 2 3
```

10.3.1.3　int[] GetGroupNumbers()

这个方法用来返回当前 Regex 对象中各捕获分组的编号，使用数字编号（int 类型变量）访

问捕获分组的速度比使用分组名（String 类型变量）要快。

```
Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
foreach (int num in regex.GetGroupNumbers()) {
Console.WriteLine(num + " ");
}
```

```
0 1 2 3
```

10.3.1.4 bool IsMatch(String input)

这个方法用来判断当前的 Regex 对象能否在字符串 *input* 中匹配某个子串。如果能则返回 true，否则返回 false。因为它并不检查整个字符串能否由正则表达式匹配，所以在数据验证时，必须在表达式首尾加上锚点\A 和\z。

```
(new Regex(@"\d")).IsMatch("a1");          // => True
(new Regex(@"\d")).IsMatch("ab");          // => False

(new Regex(@"\A\d\z")).IsMatch("a1");      // => False
(new Regex(@"\A\d\z")).IsMatch("1");       // => True
```

它有两个相应的静态便捷函数：

```
Regex.IsMatch(String input, String pattern);
Regex.IsMatch(String input, String pattern, RegexOptions options);
```

10.3.1.5 String Replace(String input, String replacement)

这个方法用来将字符串 *input* 中 Regex 对象能替换的字符串都替换为 *replacement*。

```
Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
Console.WriteLine(regex.Replace("2010-12-20 2011-02-14", "$2/$3/$1"));
```

```
12/20/2010 02/14/2011
```

在默认状态下会替换所有能匹配的子串；也可以通过设定第三个参数 *count*，人工指定替换可能发生的最多次数。

```
Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
Console.WriteLine(regex.Replace("2010-12-20 2011-02-14", "$2/$3/$1", 1));
```

```
12/20/2010 2011-02-14
```

它有三个对应的静态便捷函数：

```
Regex.Replace(String input, String pattern, String replacement);
Regex.Replace(String input, String pattern, String replacement, int count);
Regex.Replace(String input, String pattern, String replacement, RegexOptions option);
```

Replace()方法也可以不指定 *replacement* 字符串，而是提供 MatchEvaluator 对象（可以将它理解为函数指针），它接收 Match 对象，返回字符串。这样，替换操作就可以更加灵活。下面的代码展示了如何将单词统一为首字母大写格式。

```
    static string Capitalize(Match match)
    {
        //正则表达式已经用两个捕获分组分开第一个字母和之后的字母
        return match.Groups[1].Value.ToUpper() + match.Groups[2].Value.ToLower();
    }
    Console.WriteLine(new Regex(@"(?i)\b([a-z])([a-z]+)\b").Replace("one TWO tHREe",
new MatchEvaluator(Capitalize)));
    One TWO THREE
```

10.3.1.6　String[] Split(String input)

这个方法用来切分字符串 *input*，以当前的 Regex 对象匹配的子串为分隔。

```
foreach(String s in (new Regex(@"\s+")).Split("one two three")) {
 Console.WriteLine(s);
 }
one
two
three
```

在默认情况下，这个方法会切分尽可能多的次数；不过，也可以通过第二个参数 *count* 指定返回数组的大小。

```
foreach(String s in (new Regex(@"\s+")).Split("one two three", 2)) {
 Console.WriteLine(s);
 }
one
two three
```

表 10-3 总结了 *count* 取各种值的情况对结果的影响（未指定 *count* 时，返回包含 *n* 个元素的数组，切分 *n*-1 次）。

表 10–3　count 各种取值的意义

count 取值	意义
count < 0	不允许，报错
count = 0	与未设置 count 一样，切分 *n*-1 次
0 < count < *n*	返回包含 count 个元素的数组，切分 count-1 次，最后一个元素是第 count-1 次切分后右侧剩下的所有文本
count >= *n*	等于未指定 count

它有三个相应的静态便捷函数：

```
Regex.Split(String input, String pattern);
Regex.Split(String input, String pattern, int count);
Regex.Split(String input, String pattern, RegexOptions option);
```

关于匹配、替换、切分，Regex 对象都提供了对应的静态便捷函数，这样代码更加易懂，而且不用显式生成一堆对象；坏处则是每次调用都要重新检查 *pattern* 字符串，重新生成 Regex 对象。为解决这个问题，.NET 可以缓存最近的 *n* 个 Regex 对象，*n* 的默认值为 15，如果对此不满意，可以重新设置（将值设定为 0 表示禁用缓存）。

```
Regex.CacheSize = 20;
```

10.3.2 Match

Match 可以理解为“某次具体匹配的结果对象”，调用 *Match*()方法，把编译好的 Regex 对象“应用”到某个 String 对象上，就获得了作为“本次匹配结果”的 Match 对象。

每个 Match 对象对应到一次具体的匹配，Match 对象的 Success 属性标识本次匹配是否成功。如果成功，还可以通过 Index、Length、Value 三个变量获得关于此次匹配的具体信息。

Index	本次匹配文本在字符串中的开始位置
Length	本次匹配文本的长度
Value	本次匹配文本的内容

如果想进行下一次匹配，可以通过 *nextMatch*()方法获得新的 Match 对象。

```
Regex regex = new Regex(@"\d{4}-\d{2}-\d{2}");
Match match = regex.Match("2010-12-20 2011-02-14");
while(match.Success) {
 Console.WriteLine(match.Value);
 match = match.NextMatch();
}
2010-12-20
2011-02-14
```

如果正则表达式中包含捕获分组（包括普通捕获分组和命名分组），则返回的结果中会包含捕获分组匹配的信息，这些信息封装为一个 GroupCollection 对象，其中每个元素都是 Group 对象，它其实非常类似 Match 对象，也包括 Success、Value、Length、Index 等属性。如果需要某个捕获分组匹配的具体信息，可以通过直接访问对应的 Group 对象得到。

```
Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
Match match = regex.Match("2010-12-20 2011-02-14");
int[] groupNums = regex.GetGroupNumbers();
while (match.Success) {
    for (int i = 0, n = groupNums.Length; i < n; i++) {
        int groupNum = groupNums[i];
        Console.WriteLine(match.Groups[groupNum].Value + " starts at " +
            match.Groups [groupNum].Index + " and length is " +
```

```
            match.Groups[groupNum].Length);
        }
       match = match.NextMatch();
    }
    2010-12-20   starts at 0 and length is 10
    2010 starts at 0 and length is 4
    12 starts at 5 and length is 2
    20 starts at 8 and length is 2
    2011-02-14   starts at 11 and end at 21
    2011 starts at 11 and length is 4
    02 starts at 16 and length is 2
    14 starts at 19 and length is 2
```

下面是使用命名分组的程序（结果相同，略去）。

```
Regex regex = new Regex(@"(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})");
Match match = regex.Match("2010-12-20 2011-02-14");
String[] groupNames = regex.GetGroupNames();
while (match.Success) {
     for (int i = 0, n = groupNames.Length; i < n; i++) {
          String groupName = groupNames[i];
          Console.WriteLine(match.Groups[groupName].Value + " starts at " +
               match.Groups[groupName].Index + "and length is " +
               match.Groups[groupName].Length);
     }
     match = match.NextMatch();
}
```

注意：如果在 VB.NET 中访问 Match 对象的分组，不能使用 Groups[*num*]或 Groups[*name*]之类的方法，而应该使用 Groups(*num*)或 Groups(*name*)。

10.4　常用操作示例

10.4.1　验证

简单验证

```
//注意要加上\A 和\z
String regex = @"\A\d{4}-\d{2}-\d{2}\z";
Regex.IsMatch("2010-12-20", regex);  // => True
```

用 Regex 对象验证，方便反复使用

```
//同样要加上\A 和\z
Regex regex = new Regex(@"\A\d{4}-\d{2}-\d{2}\z");
String s = "2010-12-20";
regex.IsMatch(s);  // => True
```

10.4.2 提取

使用数字编号分组进行数据提取

```
Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
Match match = regex.Match("2010-12-20 2011-02-14");
int[] groupNums = regex.GetGroupNumbers();
while (match.Success) {
   Console.Write("date: " + match.Groups[0].Value);
   Console.Write(", year: " + match.Groups[1].Value);
   Console.Write(", month: " + match.Groups[2].Value);
   Console.Write(", day: " + match.Groups[3].Value);
   Console.Write("\n");
   match = match.NextMatch();
}
```

```
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

使用命名分组进行数据提取

```
Regex regex = new Regex(@"(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})");
Match match = regex.Match("2010-12-20 2011-02-14");
int[] groupNums = regex.GetGroupNumbers();
while (match.Success) {
     Console.Write("date: " + match.Groups["0"].Value);
     Console.Write(", year: " + match.Groups["year"].Value);
     Console.Write(", month: " + match.Groups["month"].Value);
     Console.Write(", day: " + match.Groups["day"].Value);
     Console.Write("\n");
     match = match.NextMatch();
}
```

```
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

10.4.3 替换

简单替换

```
String regex = @"(\d{4})-(\d{2})-(\d{2})";
String replacement = "$2/$3/$1";
String text = "2010-12-20 2011-02-14";
Console.WriteLine(Regex.Replace(text, regex, replacement));
```

```
12/20/2010 02/14/2011
```

用 Regex 对象替换，方便反复使用

```
Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");
String replacement = "$2/$3/$1";
```

```
String text = "2010-12-20 2011-02-14";
Console.WriteLine(regex.Replace(text, replacement));
12/20/2010 02/14/2011
```

使用命名分组

```
String regex = @"(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})";
String replacement = @"${month}/${day}/${year}";
String text = "2010-12-20 2011-02-14";
Console.WriteLine(Regex.Replace(text, regex, replacement));
12/20/2010 02/14/2011
```

在 replacement 中使用$

```
Regex.Replace("the price is 12.99", @"\d+\.\d{0,2}", "$$$0");
the price is $12.99
```

10.4.4　切分

简单切分

```
String regex = @"-";
foreach(String s in Regex.Split("2010-12-20", regex)) {
    Console.Write(s + " ");
}
2010 12 20
```

用 Regex 对象切分，方便重复使用

```
Regex regex = new Regex("-");
foreach(String s in regex.Split("2010-12-20")) {
    Console.Write(s + " ");
}
2010 12 20
```

第 11 章　Java

Java对正则表达式的支持比较完备，在JDK 1.4 之后，Java内置了`java.util.regex`包，提供了几乎所有常用的功能；Java 5（Java 1.5）和Java 6（Java 1.6）在正则表达式的使用上保持了一贯，主要改进是修正了少数错误。考虑到如今Java 5 是普遍使用的版本，本章的讲解以Java 5 为准。[1] 本章介绍的内容也完全适用于Android开发，Android程序中正则表达式的使用方法与本文所述的相同。

值得一提的是，Java 语言中的字符都是 Unicode 编码的，Java 的正则表达式对 Unicode 的兼容也非常好。

11.1　预备知识

Java 语言中的正则表达式的相关类都存于 `java.util.regex` 包中，如果要使用正则表达式，必须首先导入其中的类，最简单的办法是在程序文件头部写入这条语句（具体的类在后文详述）。

```
import java.util.regex.*;
```

为讲解方便，先介绍 Java 中的 `String.matches(String regex)`方法，它返回一个布尔值，表示整个 `String` 能不能由 *`regex`* 匹配。

```
"1".matches("\\d");     // => True
"1a".matches("\\d");    // => False
```

如果要检查正则表达式能否在字符串中找到匹配（也就是匹配到某个子串），可以使用下面的方法（为排版方便，这里用到了链式编程，关于 `Pattern` 的具体细节在后面详述）。

```
Pattern.compile("\\d").matcher("1a").find();// => True
```

Java 语言中的正则表达式都是以字符串的形式给出的，所以字符串转义和正则表达式转义都必须考虑。如果正则表达式中出现了反斜线`\`，在通过字符串给出时，都必须进行字符串转义，写为`\\`。比如正则表达式中的`\b`，在字符串中应该写作`\\b`，`\Q…\E` 则应当写作`\\Q…\\E`。稍微麻烦一点的是，如果要在正则表达式中使用反斜线`\`，它本身应当转义为`\\`，而在字符串中，两

[1] 更详细的文档可见于：
http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Pattern.html
http://download.oracle.com/javase/1.5.0/docs/api/java/util/regex/Matcher.html

个反斜线应当分别转义，所以字符串中应该写作\\\\。

```
//这个字符串只包含反斜线字符\
"\\".matches("\\\\");   // => True
```

11.2　正则功能详解

11.2.1　列表

表 11-1 列出了 Java 中的正则功能。

表 11–1　Java 中的正则功能列表

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	完全支持 Unicode
POSIX 字符组 ☞15	`[:digit:]`	只匹配 ASCII 字符，写成`\P{…}`
Unicode 属性 ☞127	`\p{…}`	完全支持 Unicode
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	采用 ASCII 匹配规则
行起始位置 ☞62	`^` `\A`	支持
行结束位置 ☞62	`$` `\z` `\Z`	支持
单词边界 ☞123	`\b` `\B`	采用 Unicode 规则
顺序环视 ☞69	`(?=…)` `(?!…)`	支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	不支持无法确定长度范围的表达式
匹配模式 ☞83	`i m s x`	支持
模式作用范围 ☞91	`(?`*modifier*`)` `(?-`*modifier*`)`	支持
纯文本模式 ☞101	`\Q…\E`	支持(存在小问题，到 Java 6 才修正)
捕获分组及引用☞44	`(…)` `\`*num* **$***num*	支持
命名分组 ☞53	`(?<`*name*`>…)` `\k{`*name*`}` `${`*name*`}`	Java 7 以上支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

11.2.2　字符组

在 Java 语言中所有的字符都采用 UTF-16 编码，所以在字符组中可以直接使用中文，因而不会出现多字节字符错误匹配的问题，而且点号`.`可以匹配中文字符。

```
"你".matches(".");           // => True
"遭".matches("[正则]");      // => False
```

值得注意的一点是，Java 语言中的字符组可以进行集合运算，通常主要用于字符组的“减”运算——如果需要匹配英文中所有小写的辅音字母，也就是从 26 个小写字母中“减去”5 个元音字母，可以写作`[[a-z]&&[^aeiou]]`。它的意思是，“从 a 到 z 的所有字符（也就是小写英文字符）”与“除 a、e、i、o、u 之外的任何字符”取交集，也就是所有的小写辅音字母。

```
"a".matches("[[a-z]&&[^aeiou]]");     // => False
"b".matches("[[a-z]&&[^aeiou]]");     // => True
```

也可以用这个功能实现匹配“英文大小写字母和数字（不包括下画线）”。

```
"a".matches("[\\w&&[^_]]"); // => True
"0".matches("[\\w&&[^_]]"); // => True
"_".matches("[\\w&&[^_]]"); // => False
```

相对于`&&`的“交”运算，Java 也支持在字符组内用`||`进行“并”运算，不过因为“并集”的意思就是“给字符组内部添加字符”，所以并不需要设定特殊的运算符，比如`[[0-4]||[6-9]]`就等价于`[[0-4][6-9]]`，也等价于`[0-46-9]`。虽然这几种写法都没错，但最后那种写法显然更简单，所以我推荐最后的写法。

```
"3".matches("[[0-4][6-9]]"); // => True
"5".matches("[[0-4][6-9]]"); // => False
```

因为“并”是默认的运算，如果字符组内部只出现一个字符组，可以理解为“单个字符组与空集取并集”，换句话说，`[[0-4]]`等价于`[0-4]`。看到`[[0-4]]`，而不是包含 7 个字符（`0`、`1`、`2`、`3`、`4`、`[`、`]`）的字符组。

```
"3".matches("[[0-4]]"); // => True
"[".matches("[[0-4]]"); // => False
```

Java 中可以用`\u`*hex* 指定 Unicode 码值，其中 *hex* 为 4 位十六进制数，所以在 Unicode 编码环境下可以用字符组`[\u4E00-\u9FFF]`匹配所有的中文字符。

```
"我".matches("[\u4E00-\u9FFF]"); // => True
```

Java 中也可以使用 POSIX 字符组`[:`*name*`:]`，不过必须写为`\p{`*name*`}`，而且第一个字母必须大写，比如`[:digit:]`就应该写成`\p{Digit}`，其他 POSIX 字符组都是如此，只有`[:xdigit:]`必须写成`\p{XDigit}`。注意，所有的 POSIX 字符组都只能匹配 ASCII 字符。

```
"0".matches("\\p{Digit}"); // => True
"f".matches("\\p{XDigit}"); // => True
```

11.2.3　Unicode 属性

Java 对 Unicode 字符组的支持比较好，它支持 Unicode Property。这样，用`\p{N}`就可以匹配所有的数字字符，包括中文的全角数字字符，比如０、１、２等；`\p{P}`可以匹配各种标点符号，包括中文的冒号：等。

```
"１".matches("\\p{N}"); // => True
"：".matches("\\p{P}"); // => True
```

Java也支持Unicode Block，其记法是以In为前缀的，比如匹配中文字符的Unicode Block就应当写为`InCJK_Compatibility_Ideographs`，这与.NET不一样，使用时应注意。[1]

```
"我".matches("\\p{InCJK_Compatibility_Ideographs}"); // => True
```

关于 Unicode 属性的细节，请参考第 130 页。

11.2.4　字符组简记法

在 Java 的正则表达式中，`\d`、`\D`、`\w`、`\W`、`\s`、`\S`都使用 ASCII 匹配规则。也就是说，`\d`等价于`[0-9]`，`\w`等价于`[0-9a-zA-Z_]`，`\s`无法匹配 ASCII 编码之外的空白字符。

```
//半角字符
"1".matches("\\d"); // => True
"a".matches("\\w"); // => True
" ".matches("\\s"); // => True
//全角字符及中文字符
"１".matches("\\d"); // => False
"我".matches("\\w"); // => False
"　".matches("\\s"); // => False
```

11.2.5　单词边界

Java 正则表达式中的单词边界有点特殊：一般来说，`\b`的匹配受到`\w`的影响，但 Java 却不是如此（☞122）。尽管`\w`使用 ASCII 规则，但`\b`定义的“单词字符”是遵循 Unicode 规则的。

这一点官方文档并没有详细解释，不过我们可以这样粗略记忆：`\b`所识别的单词字符，可以是各语言中的单词字符（包括中文字符），但不包括空白和标点（包括中文的空白和标点）；而且，单词字符必须出现。

```
//英文字符-半角标点
Pattern.compile("a\\b").matcher("a,").find();       // => True
```

[1] Unicode Block 的细节可参考 http://download.oracle.com/javase/6/docs/api/java/lang/Character.UnicodeBlock.html。

```
//英文字符-全角标点
Pattern.compile("a\\b").matcher("a，").find();      // => True
//英文字符-结束
Pattern.compile("a\\b").matcher("a ").find();       // => True
//中文字符-半角标点
Pattern.compile("我\\b").matcher("我,").find();     // => True
//中文字符-全角标点
Pattern.compile("我\\b").matcher("我，").find();    // => True
//中文字符-空字符串
Pattern.compile("我\\b").matcher("我").find();      // => True
//全角标点-结束（未出现单词字符）
Pattern.compile("，\\b").matcher("，").find();      // => False
//中文字符-英文字符（未出现单词字符）
Pattern.compile("我\\b").matcher("我a").find();     // => False
```

11.2.6 行起始/结束位置

^匹配的是“行开始的位置”，在默认情况下它只能匹配“整个字符串的开始位置”，如果指定使用多行模式（Multiline Mode），^可以匹配字符串内部的文本行的开始位置；而\A无论在什么情况下都匹配“整个字符串的开始位置”。

```
Pattern.compile("^1").matcher("1\n2\n").find();     // => True
Pattern.compile("^2").matcher("1\n2\n").find();     // => False
Pattern.compile("(?m)^2").matcher("1\n2\n").find(); // => True

Pattern.compile("\\A1").matcher("1\n2\n").find();   // => True
Pattern.compile("\\A2").matcher("1\n2\n").find();   // => False
Pattern.compile("(?m)\\A2").matcher("1\n2\n").find();    // => False
```

在默认情况下，Java 中的$、\Z、\z 匹配的是整个字符串的结束位置（如果结束位置有换行符，则$和\Z 匹配这个换行符之前的位置），多行模式只会影响到$的匹配，这时候它可以匹配文本内部行的结束位置，所以进行准确的验证时应当使用\z。

```
Pattern.compile("1$").matcher("1\n").find();        // => True
Pattern.compile("1$").matcher("1\n2").find();       // => False
Pattern.compile("(?m)1$").matcher("1\n2").find();   // => True

Pattern.compile("1\\Z").matcher("1\n").find();      // => True
Pattern.compile("1\\Z").matcher("1\n2").find();     // => False
Pattern.compile("(?m)1\\Z").matcher("1\n").find();  // => True

Pattern.compile("1\\z").matcher("1").find();        // => True
Pattern.compile("1\\z").matcher("1\n").find();      // => False
Pattern.compile("(?m)1\\z").matcher("1\n").find();  // => False
```

11.2.7　环视

在 Java 中的肯定顺序环视`(?=…)`和否定顺序环视`(?!…)`内可以使用任意形式的子表达式。

```
"cab".matches("c(?=ab+)");            // => True
"ccd".matches("c(?=(ab+|cd))");       // => True
"cabbbb".matches("c(?=(ab+|cd))");    // => True
```

逆序环视中使用的子表达式**必须有长度上限**。也就是说，`(?<=ab)`、`(?<=ab?)`、`(?<=ab{4,12})`都是合法的，`(?<=ab+)`却会报错。这个问题值得注意，因为它在代码编译阶段不会抛出异常，而在运行时会出现。

```
"ab".matches("ab(?<=ab)");            // => True
"cd".matches("cd+(?<=(abcd|cd))");    // => True
"cd".matches("cd(?<=(ab+|cd))");      //编译时没有问题，运行却会报错
```

11.2.8　匹配模式

表 11-2 列出了 Java 中的正则表达式支持的匹配模式。

表 11–2　Java 中的正则表达式支持的匹配模式

常量	修饰符	说明
CASE_INSENSITIVE	i	不区分 ASCII 字符的大小写
COMMENTS	x	允许正则表达式中出现注释，表达式中的空白字符，以及#开始到行末的文本，都视为注释
MULTILINE	m	允许^和$不仅匹配字符串的起始和结束位置，还可以匹配字符串内部文本行的起始和结束位置
DOTALL	s	允许点号.匹配任何字符，包括换行符
UNICODE_CASE	u	可以识别 Unicode 字符的不同形态，“不区分大小写”的范围不限于 ASCII 字符，但严重影响性能
UNIX_LINES	d	限定.、^、$能识别的行终结符只有换行符\n，忽略\r\n 和 Unicode 行终结符等其他字符
CANON_EQ	无	匹配时采取 Unicode“等价”规则，可以识别意义相等的复合字符（单个字符加上调号）与单个字符；但此选项会严重影响性能

匹配模式可以在表达式中以`(?modifier)`指定，也可以在生成 Pattern 对象时，以预定义常量指定。

```
"A".matches("a");       // => False
"A".matches("(?i)a");   // => True
Pattern.compile("a", Pattern.CASE_INSENSITIVE).matcher("A").find();   // => True
```

Java 也支持用`(?-modifier)`停用某个匹配模式。

```
"aBb".matches("a(?i)b(?-i)b");    // => True
"aBB".matches("a(?i)b(?-i)b");    // => False
```

11.2.9 纯文本模式

在`\Q`和`\E`之内，所有的元字符和特殊结构都失去特殊意义，只能匹配它们对应的字符本身。不过在字符串中，`\Q`和`\E`中的反斜线也需要经过字符串转义，应该写成`\\Q`和`\\E`。[1]

```
"A".matches("\\Q.\\E");          // => False
".".matches("\\Q.\\E");          // => True
"[a-f]".matches("\\Q[a-f]\\E");  // => True
"b".matches("\\Q[a-f]\\E");      // => False
```

11.2.10 捕获分组的引用

在 Java 中，如果要在正则表达式内部引用捕获分组，应当使用`\num`记法，其中 *num* 为对应捕获分组的编号。

```
"ab".matches("([a-z])\\1");      // => False
"aa".matches("([a-z])\\1");      // => True
```

要在替换时引用捕获分组，应当使用`$num`记法。

```
"2010-12-20".replaceAll("(\\d{4})-(\\d{2})-(\\d{2})", "$2/$3/$1");
12/20/2010
```

要在 *replacement* 字符串中表示`$`字符，必须使用`\$`转义。

```
"the price is 12.99".replaceAll("\\d+\\.\\d{0,2}", "\\$$0");
the price is $12.99
```

11.3 正则 API 简介

Java 中正则表达式相关的类都在 `java.util.regex` 之内，一般来说，主要用到的是这两个类：`java.util.regex.Pattern`（以下简称 `Pattern`）和 `java.util.regex.Matcher`（以下简称 `Matcher`）。`Pattern` 对应正则表达式，一个 `Pattern` 与一个 `String` 对象关联，生成一个 `Matcher`，它对应 `Pattern` 在 `String` 中的一次匹配；调用 `Matcher` 对象的 *`find()`*方法，`Matcher` 对象就会更新为下一次匹配的匹配信息。`Pattern`、`String`、`Matcher` 对象之间的联系如图 11-1 所示，代码如例 11-1 所示。

[1] 但是此功能在 Java 5 中有 bug，如果在字符组内部使用，可能会出现奇怪的问题，到了 Java 6 才可以彻底解决。

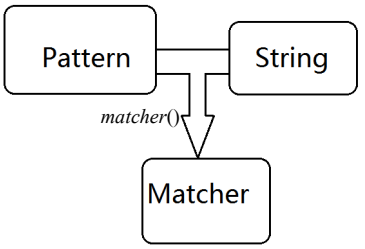


图 11–1　Java 中与正则表达式相关的类

例 11–1　Java 中正则表达式相关类的实际使用

```
Pattern pattern = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
String string = "2010-12-20 2011-02-14";
Matcher matcher = pattern.matcher(string);
while(matcher.find()) {
 System.out.println(matcher.group(0));
}
```

```
2010-12-20
2011-02-14
```

11.3.1　Pattern

Pattern 是 Java 语言中的正则表达式对象（字符串与正则表达式的差别详见第 8 章）。要使用正则表达式，必须首先从字符串“编译”出 Pattern 对象，这需要用到 Pattern.compile(String *regex*)方法。

```
Pattern pattern = Patterm.compile("a.b+");
```

如果要指定匹配模式，可以在表达式中使用`(?modifier)`修饰符指定，也可以使用预定义常量。下面的两个 Pattern 对象的生成方法不同，结果却是等价的。

```
Pattern pattern = Patterm.compile("(?i)a.b+");
Pattern pattern = Patterm.compile("a.b+", Pattern.CASE_INSENSITIVE);
```

如果要同时指定多种模式，可以连写模式修饰符，也可以直接用|运算符将预定义常量连接起来，以下两个 Pattern 对象也是等价的。

```
Pattern pattern = Patterm.compile("(?is)a.b+");
Pattern pattern = Pattern.compile("a.b+", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
```

下面介绍 Pattern 的主要成员方法。

11.3.1.1 static boolean matches(String regex, CharSequence input)

这个方法可以检验字符串 *input* 能否由正则表达式 *regex* 匹配，因为是静态方法，所以不需要编译生成各个对象，方便“随手”使用。要注意的是，它检验的是“整个字符串能否由表达式匹配”，而不是“表达式能否在字符串中找到匹配”。你可以认为 *regex* 的首尾自动加上了匹配字符串起始和结束位置的锚点`\A`和`\z`。

```
Pattern.matches("\\d{6}", "a123456"); // => False
Pattern.matches("\\d{6}", "123456");  // => True
```

11.3.1.2 String[] split(CharSequence text)

通常，`Pattern` 对象需要配合下面将要介绍的 `Matcher` 一起完成正则操作。如果只用正则表达式来切分字符串，只用 `Pattern` 的这个方法也可以。

这个方法接收的参数类型是 `CharSequence`，你对它可能有点陌生，其实它是 `String` 的父类，其他子类还有 `CharBuffer`、`StringBuffer`、`StringBuilder`，因而可以应对常见的各种表示“字符串”的类。下面的代码仅以 `String` 为例。

```
String s = "2010-12-20-";
Pattern pattern = Pattern.compile("\\s+");
for(String part : pattern.split(s)) {
 System.out.println(part);
}
2010
12
20
```

请注意，虽然正则表达式能在字符串中匹配 3 次，但切分得到的结果数是 3 个而不是 4 个，因为末尾空白字符之后的空字符串会被自动忽略。如果这个空字符串出现在其他位置（比如字符串开头或字符串中），则会被保留。

11.3.1.3 String[] split(CharSequence text, int limit)

这个方法与上面介绍的方法很相似，只是多了一个参数 *limit*，它用来限定返回的 `String` 数组的最大长度。也就是说，它规定了字符串至多只能“切” *limit*-1 次。如果切分前明确指定了切分的次数或者返回结果的元素数，就可以使用这个方法。

```
String s = " 2010-12-20 ";
Pattern pattern = Pattern.compile("\\s+");
for(String part : pattern.split(s, 2)) {
 System.out.println(part);
}
2010
12-20
```

既然 *limit* 是 `int` 类型，那么它自然可以设定为各种值，表 11-3 总结了 *limit* 在各个取值

区间对结果的影响（未指定 *limit* 时，最终返回包含 *n* 个元素的数组，实际能切分的次数是 *n*-1）。

表 11–3　limit 取值对结果的影响

取值	结果
limit<0	等于未设定 limit 时，保留末尾的空字符串
limit=0	等于未设定 limit 时，切分 *n*-1 次，忽略末尾的空字符串
0 <limit<n	返回数组包含 limit 个元素，切分 limit-1 次，最后一个元素是第 limit-1 次切分后，右侧剩下的所有文本
limit>=n	效果与未指定 limit 相同

11.3.1.4　static String quote(String text)

这个方法用来取消字符串 *text* 中所有转义字符的特殊含义，等价于在字符串首尾添加\Q和\E。通常，如果需要把某个字符串作为没有任何特殊意义的正则表达式（比如从外界读入的字符串，用在某个复杂的正则表达式中），就可以使用这个方法。

```
"aacb".matches("a*.b"); // => True
"a*.b".matches("a*.b"); // => False
"a*.b".matches("a*.b"); // => False
"a*.b".matches(Pattern.quote("a*.b")); // => True
```

11.3.2　Matcher

Matcher 可以理解为“某次具体匹配的结果对象”：把编译好的 Pattern 对象“应用”到某个 String 对象上，就获得了作为“本次匹配结果”的 Matcher 对象。之后，就可以通过它获得关于匹配的信息。

```
Pattern pattern = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
while(matcher.find()) {
     System.out.println(matcher.group());
}
2010-12-20
2011-02-14
```

对编译好的 Pattern 对象调用 matcher(String *text*)方法，传入要匹配的字符串 *text*，就得到了 Matcher 对象，每调用一次 *find*()方法，如果返回 true，就表示“找到一个匹配”，此时可以通过下面的若干方法获得关于本次匹配的信息。

11.3.2.1　String group(int n)

返回当前匹配中编号为 *n* 的捕获分组匹配到的文本，如果 *n* 为 0，则取匹配的全部内容；如果 *n* 小于 0 或者大于最大分组编号数，则报错。

11.3.2.2 String group()

返回当前匹配的全部文本，相当于 *group*(0)。

11.3.2.3 int groupCount()

返回此 Matcher 对应 Pattern 对象中包含的捕获分组数目，编号为 0 的默认分组不计在内。

11.3.2.4 int start(n)

返回当前匹配中第 *n* 对捕获括号匹配的文本在原字符串中的起始位置。

11.3.2.5 int start()

返回当前匹配的文本在原字符串中的起始位置，相当于 *start*(0)。

11.3.2.6 int end(n)

返回当前匹配中第 *n* 对捕获括号匹配的文本在原字符串中的结束位置。

11.3.2.7 int end()

返回当前匹配的文本在原字符串中的结束位置，相当于 *end*(0)。

下面的例子集中示范了这几个方法。

```
Pattern pattern = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
while(matcher.find()) {
    System.out.println(matcher.group() + " starts at " + matcher.start() + " and end
at " + matcher.end());
 for (int i = 0, n = matcher.groupCount(); i <= n; i++) {
     //注意这里是 i<=n，因为编号为 n 的分组一定存在
     System.out.println(matcher.group(i) + "  starts at " + matcher.start(i) + " and
ends at " + matcher.end(i));
 }
}
```

```
2010-12-20 starts at 0 and ends at 10
2010-12-20 starts at 0 and ends at 10
2010 starts at 0 and ends at 4
12 starts at 5 and ends at 7
20 starts at 8 and ends at 10
2011-02-14 starts at 11 ands end at 21
2011 starts at 11 and ends at 15
02 starts at 16 and ends at 18
14 starts at 19 and ends at 21
```

11.3.2.8　String replaceAll(String replacement)

如果进行正则表达式替换，一般用到的是 Matcher 的 ***replaceAll()***方法，它会将原有文本中正则表达式能匹配的所有文本替换为 ***replacement*** 字符串。

```
Pattern pattern = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
System.out.println(matcher.replaceAll("Date"));
```

```
Date Date
```

如果要在 ***replacement*** 字符串中引用之前某个捕获分组匹配的文本，则应当使用**$*num***，其中 ***num*** 为捕获分组的编号。

```
Pattern pattern = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
System.out.println(matcher.replaceAll("$2/$3/$1"));
```

```
12/20/2010 02/14/2011
```

如果要在 ***replacement*** 中表示**$**符号，则应当将它转义为**\$**。

```
Pattern pattern = Pattern.compile("(\\d+\\.\\d+");
Matcher matcher = pattern.matcher("the price is 12.99");
System.out.println(matcher.replaceAll("\\$$0"));
```

```
the price is $12.99
```

11.3.2.9　Matcher appendReplacement(StringBuffer sb, String replacement)

某些时候，我们并不希望“一次性”进行所有的正则表达式替换，而是需要更细致的控制能力，这时候就应当用到 appendReplacement 方法。它的目的很清楚：从上一次匹配结束的位置开始（如果这是第一次匹配，就从字符串首部开始），找到最近的匹配，得到本次匹配起始位置之前的文本+本次匹配替换之后的文本。这个方法有一点怪异，因为匹配结果并不会作为返回参数，而是附加到作为输入参数的 StringBuffer 中，返回的是“已经更新到最近一次匹配状态”的 Matcher。下面的代码显示了匹配的过程。

```
Pattern numbers = Pattern.compile("\\d+(?:\\.\\d+)");
Matcher m = numbers.matcher("Apple costs 3.01, orange costs 2.58, pear costs 5.74");
StringBuffer sb = new StringBuffer();
while (m.find()) {
    m.appendReplacement(sb, "\\$$0");
    System.out.println(sb.toString());
}
System.out.println(sb.toString());
```

```
Apple costs $3.01
Apple costs $3.01, orange costs $2.58
Apple costs $3.01, orange costs $2.58, pear costs $5.74
```

11.3.2.10 Matcher appendTail(StringBuffer sb)

这个方法大多数情况下是和上面的 appendReplacement 配合使用的。因为 appendReplacement 能调用的前提是“剩下的文本中仍然存在匹配”，如果这个前提不存在，就无法取到剩余文本了。appendTail 用来把“最后一次匹配成功的结束位置到字符串结束位置的文本”全部附加到 StringBuffer 中。

```
Pattern numbers = Pattern.compile("\\d+(?:\\.\\d+)");
Matcher m = numbers.matcher("Apple costs 3.01, orange costs 2.58, expensive!");
StringBuffer sb = new StringBuffer();
while (m.find()) {
    m.appendReplacement(sb, "\\$$0");
}
System.out.println(sb.toString());
m.appendTail(sb);
System.out.println(sb.toString());

Apple costs $3.01, orange costs $2.58
Apple costs $3.01, orange costs $2.58, expensive!
```

11.3.3 String

许多时候只需要临时使用某个正则表达式，而不需要重复使用，这时候每次都生成 Pattern 对象和 Matcher 对象再操作显得很烦琐。所以，Java 的 String 类提供了正则表达式操作的静态成员方法，只需要 String 对象就可以执行正则表达式操作。下面讲解 String 类常见的正则操作。

11.3.3.1 boolean matches(String regex)

这个方法判断当前的 String 对象能否由正则表达式 *regex* 匹配。请注意，这里的“匹配”指的并不是 *regex* 能否在 String 内找到匹配，而是指 *regex* 匹配整个 String 对象，因此非常适合用来做数据校验。

```
"123456".matches("\\d{6}");       // => True
"a123456".matches("\\d{6}");      // => False
```

11.3.3.2 String replaceFirst(String regex, String replacement)

这个方法用来替换正则表达式 *regex* 在字符串中第一次能匹配的文本，可以在 *replacement* 字符串中用*$num* 引用 *regex* 中对应捕获分组匹配的文本。

```
"2010-12-20 2011-02-14".replaceFirst("\\d{4}-\\d{2}-\\d{2}", "Date");
Date 2010-02-14

"2010-12-20 2011-02-14".replaceFirst("(\\d{4})-(\\d{2})-(\\d{2})", "$2/$3/$1");
12/20/2010 2011-02-14
```

11.3.3.3　String replaceAll(String regex, String replacement)

这个方法用来进行所有的替换，它的结果等同于 Matcher 类的 replaceAll()方法，*replacement* 字符串中也可以用*$num* 的表示法引用 *regex* 中对应捕获分组匹配的文本。

```
"2010-12-20 2011-02-14".replaceAll("\\d{4}-\\d{2}-\\d{2}", "Date");
Date Date

"2010-12-20 2011-02-14".replaceAll("(\\d{4})-(\\d{2})-(\\d{2})", "$2/$3/$1");
12/20/2010 02/14/2011
```

值得注意的是，在 Java 5.0 之前的版本中，String 类并没有提供简单的“字符串替换”方法，只有 replaceAll()。如果你使用的 Java 版本低于 5.0，又需要进行简单的字符串替换，可以使用 replaceAll()，不过应该在要被替换的字符串首尾加上\Q 和\E。

11.3.3.4　String[] split(String regex)

这个方法等价于 Pattern 中对应的 *split*()方法，此处不再赘述。

11.3.3.5　String[] split(String regex, int limit)

这个方法等价于 Pattern 中对应的 *split()*方法，此处不再赘述。

String 类的这些方法，其实都是包装了 Pattern 对象和 Matcher 对象，只是操作更方便，并不能提高运行的效率。

11.4　常用操作示例

11.4.1　验证

简单验证

```
"2010-12-20".matches("\\d{4}-\\d{2}-\\d{2}");          // => True
Pattern.matches("\\d{4}-\\d{2}-\\d{2}", "2010-12-20"); // => True
```

使用 Pattern 对象，方便反复验证

```
//注意要加上\A 和\z
Pattern pattern = Pattern.compile("\\A\\d{4}-\\d{2}-\\d{2}\\z");
String s = "2010-12-20";
pattern.matcher(s).find();// => True
```

11.4.2　提取

```
Pattern pattern = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
```

```
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
while(matcher.find()) {
    System.out.println(matcher.group());
}
2010-12-20
2011-02-14

Pattern pattern = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
while(matcher.find()) {
    System.out.print("date: " + matcher.group(0));
    System.out.print(", year: " + matcher.group(1));
    System.out.print(", month: " + matcher.group(2));
    System.out.print(", day: " + matcher.group(3));
    System.out.print("\n");
}
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

11.4.3 替换

简单替换

```
String regex = "(\\d{4})-(\\d{2})-(\\d{2})";
String replacement = "$2/$3/$1";
"2010-12-20 2011-02-14".replaceAll(regex, replacement);
12/20/2010 02/14/2011
```

使用 Pattern 对象，方便反复使用

```
Pattern pattern = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");
Matcher matcher = pattern.matcher("2010-12-20 2011-02-14");
System.out.println(matcher.replaceAll("$2/$3/$1"));
12/20/2010 02/14/2011
```

在 replacement 中使用$

```
String regex = "\\d+\\.\\d{0,2}";
String replacement = "\\$$0";
System.out.println("price is 12.99".replaceAll(regex, replacement));
price is $12.99
```

11.4.4　切分

简单切分

```
String regex = "\\s+";
for(String s : "one\ttwo\n three".split(regex)) {
    System.out.print(s + " ");
}
one two three
```

使用 Pattern 对象，方便反复使用

```
Pattern pattern = Pattern.compile("\\s+");
for(String s : pattern.split("one\ttwo\n three")) {
     System.out.println(s);
}
one two three
```

11.5　Java 8 和 Java 9 的新改进

从 2012 年本书第 1 版出版以来，Java 语言经历了多次重大更新，与正则表达式相关的功能也有所变化，下面总结了 Java 8 和 Java 9 中正则表达式相关的变化。

11.5.1　Java 8 的新改进

改进 1：新增转义序列`\h`，表示 Unicode 标准中定义的任何水平方向的空白字符。

改进 2：在 Java 4 到 Java 7 中，`\v` 只能匹配垂直制表符（Vertical Tab），到了 Java 8，`\v` 扩大为能匹配任何垂直空白字符，不再局限于垂直制表符，如果一定要匹配垂直制表符，请使用`\x0B` 或者`\cK`。

改进 3：Java 8 新增了`\R`，用来匹配Unicode标准中定义的“换行符”，既可以匹配`\r`，又可以匹配`\n`，还可以匹配`\r\n`，但是不能匹配`\n\r`，也不能把`\r\n`中的`\r`或者`\n`割裂出来匹配。因为目前普通人并不会刻意区分“回车”和“换行”，更习惯把它们视为“一个字符”，所以`\R`使用起来无疑更加方便。请注意，它等同于`(?>\r\n|[\n\cK\f\r\u0085\u2028\u2029])`，其中包含固化分组 [1]，所以`\R`并不是一个普通的“字符组简记法”，不能用在字符组中。

[1] 简单来说，固化分组（Atomic Group）是这样的捕获分组：在进行匹配尝试时，一旦其中的表达式匹配了文本，就会“固化”下来不放弃，哪怕完整的表达式因此匹配失败也无所谓。具体细节请参考《精通正则表达式》一书中关于固化分组的讲解。

11.5.2 Java 9的新改进

在Java 9中，正则表达式的改进全部发生在Matcher类中，主要分为三类。

第一类，重载了原有方法，提高效率。具体是appendReplacement和appendTail方法，之前只能接收StringBuffer，现在可以接收StringBuilder。熟悉Java语言的读者都知道，如果不涉及多线程，StringBuilder的效率高于StringBuffer。因为之前已经讲解过这两个方法，所以这里不赘述。大家使用时直接使用StringBuilder类型的参数就可以了。

第二类，也是重载了原有方法，让字符串处理变得更加灵活。replaceAll和replaceFirst两个方法，原来的第二个参数只能是String类型，现在可以使用Function<MatchResult, String>，提供了对匹配结果进行细致而复杂的操作。比如像下面这样，找到所有首字母大写的单词，将整个单词全部转换为大写。

```
Pattern pattern = Pattern.compile("\\b[A-Z][a-z]+\\b");
Matcher matcher = pattern.matcher("I need Someone to READ The BOok");
System.out.println(matcher.replaceAll(mr -> mr.group().toUpperCase()));
I need SOMEONE to READ THE BOok
```

第三类，是顺应Java新提供的函数式编程范式，提供了流式处理的能力。对Matcher对象调用results()方法，可以获得一个由MatchResult构成的stream，然后可以直接进行函数式编程。下面的代码实现了从一段文本中找出所有数值并求和的功能。

```
Pattern prices = Pattern.compile("\\d+(?:\\.\\d+)");
Matcher m = prices.matcher("Apple costs 3.01, orange costs 2.58, pear costs 5.74");
Double total = m.results().mapToDouble(x -> Double.parseDouble(x.group())).sum();
System.out.println("total cost: " + total);
total cost: 11.33
```

第 12 章　JavaScript

JavaScript 中正则表达式的情况比较复杂，因为同样的 JavaScript 代码在不同浏览器上的表现可能不同，幸运的是，本章介绍的功能几乎都是“通用”的，特别情况会专门标注出来。

JavaScript 1.2（发布于 1997 年）以后的版本都内建了正则表达式支持，从 JavaScript 1.5（发布于 2000 年 11 月）开始，对正则表达式的支持更加完备了。JavaScript 1.5 的规范对应于 ECMA-262 Edition 3 的标准，目前流行的所有主流浏览器（包括 IE、Firefox、Chrome、Safari 等）都支持此标准。所以通常来说，JavaScript 的正则表达式在各种浏览器中的表现都是相同的，但是也有例外。如果同样的表达式在不同的浏览器中表现不同，本章会注明。

自从本书初版面世以来，ECMAScript 又更新了若干版本，其中 2015 年发布的 ES2015（ES6）和 2017 年发布的 ES2017（技术委员会为 TC39）中，都有与正则表达式相关的改进。考虑到本修订版的出版时间，主流的浏览器中应当都可以直接使用这些新功能了。不过为谨慎起见，涉及这些改进的地方，本章都会注明。

12.1　预备知识

JavaScript中的正则表达式是`RegExp`。[1] 最常见的生成方法是在首尾两个斜线分隔符`/`之间，用正则文字给出表达式，比如`/\d+/`。

为了讲解方便，先介绍 `RegExp` 的 `test()`实例方法：`RegExp.prototype.test(`*`string`*`)`返回一个布尔值，表示表达式能否在参数 *`string`* 中找到匹配。注意它并不要求整个 *`string`* 由表达式匹配，如果要检验整个 *`string`* 能否由表达式匹配，可以在表达式两端添加`^`和`$`，因为 JavaScript 不支持`\A`、`\Z`、`\z`。

```
/\d/.test("1");         // => true
/\d/.test("a1");        // => true
/^\d$/.test("a1");      // => false
```

[1] 如果你用过 Ruby 或是 Golang 的正则表达式，可能会发现 Ruby、Golang、JavaScript 中的正则表达式名字很像，唯一的区别在于大小写：Ruby 中是 Regexp，Golang 中是 regexp，JavaScript 中是 RegExp。

RegExp 也可以显式生成，可以使用的参数有两种类型：一种是使用字符串文字，这时需要同时考虑字符串转义和正则表达式转义；另一种是使用正则文字，直接在表达式两端写上反斜线/作为分隔符即可。如果使用正则文字就不需要考虑字符串的转义（但是别忘了正则表达式元字符的转义 ☞8），唯一需要特殊考虑的字符就是反斜线/，如果它在正则表达式内部出现，必须转义为\/。

```
(new RegExp("\\d")).test("1");  // => true
(new RegExp(/\d/)).test("1");   // => true
(new RegExp(/\//)).test("/");   // => true
```

相比起来，正则文字的方式更直观，但是既然已经有了正则文字，再显式创建 RegExp 就显得多此一举。而字符串文字虽然看起来麻烦点，却可以直接对接其他字符串变量，在程序中使用起来反而更加灵活。

12.2 正则功能详解

12.2.1 列表

表 12-1 中列出了 JavaScript 中的正则功能。

表 12–1 JavaScript 中的正则功能

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	可以使用 Unicode 字符
POSIX 字符组 ☞15	`[:digit:]`	不支持
Unicode 属性 ☞127	`\p{…}`	ES2017（TC39）之后支持
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	部分采用 ASCII 匹配规则
行起始位置 ☞62	`^`	只支持`^`
行结束位置 ☞62	`$`	只支持`$`
单词边界 ☞123	`\b` `\B`	采用 ASCII 匹配规则
顺序环视 ☞69	`(?=…)` `(?!…)`	支持（限 JavaScript 1.5 以后）
逆序环视 ☞69	`(?<=…)` `(?<!…)`	ES2017（TC39）之后支持
匹配模式 ☞83	`i` `m` `s` `g`	常用模式中只支持 `i` 和 `m`，ES2017 支持 `s`，`g` 模式在下文详细讲解
模式作用范围 ☞91	`(?`*modifier*`)` `(?-`*modifier*`)`	不支持
纯文本模式 ☞101	`\Q…\E`	不支持
捕获分组及引用☞44	`(…)` `\`*num* `$`*num*	支持

（续表）

功能	记法	说明
命名分组 ☞53	`(?<name>…)` `\k{name}` `${name}`	ES2017（TC39）之后支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

12.2.2　字符组

在 JavaScript 的正则表达式中，中文字符问题比较复杂，如果 JavaScript 程序使用的是非 Unicode 编码（比如常见的 GBK 编码），在某些浏览器中，下面的程序可能返回 `true`（具体取决于所使用的浏览器）。

```
/[正则]/.test("遭");     // => true
```

解决办法是使用 Unicode 编码环境，这样正则表达式不会将多字节字符拆成“字节”来对待，可以避免错误匹配。其中具体的细节，请参考第 116 页的讲解。

```
/[正则]/.test("遭");     // => false
```

在 JavaScript 中可以用`\u`*hex* 指定 Unicode 码值，其中 *hex* 为码值的十六进制表示。所以，在 Unicode 编码环境下可以用字符组`[\u4E00-\u9FFF]`匹配所有的中文字符。

```
/^[\u4E00-\u9FFF]$/.test("遭");  // => true
/^[\u4E00-\u9FFF]$/.test("a");   // => false
```

根据 ES2017（TC39），JavaScript 中提供了全新的指定 Unicode 字符的方式，也就是用`\u{`*hex*`}` 指定 Unicode 码值，其中 *hex* 为码值的十六进制表示，但不限于 4 位十六进制（可以参考第 7 章内容）。同时必须显式指定正则表达式为 Unicode 模式（参考下文）。所以，我推荐采用更新的写法 `[\u{4E00}-\u{9FFF}]`匹配所有的中文字符。

```
/^[\u{4E00}-\u{9FFF}]$/u.test("遭"); // => true
/^[\u{4E00}-\u{9FFF}]$/u.test("a");  // => false
```

要注意的是，如果用点号`.`匹配多字节字符，在不同浏览器中的表现是不一样的，有时候可能匹配单个字节，有时候可能匹配整个多字节字符。在笔者的计算机中，同一段 JavaScript 代码在 Firefox 和 Chrome 中运行的结果不同。

Firefox

```
/^.$/.test("遭");  // => true
```

Chrome

```
/^.$/.test("遭");  // => false
```

不过，2015 年发布的 ES2015（ECMAScript 6）很好地解决了这个问题，它提供了匹配模式 u，u 会影响点号.对多字节字符的匹配，在不同浏览器中的表现完全相同。如果不是受到客观条件的限制，我建议在 JavaScript 中所有用到正则表达式的地方，都指定匹配模式 u。本章其余代码都会照这样操作。

```
/^.$/u.test("遭"); // => true
```

12.2.3 字符组简记法

在 JavaScript 的正则表达式中，\d、\D、\w、\W 都采用 ASCII 匹配规则，即便在 Unicode 编码环境下仍是如此。也就是说，\d 等价于[0-9]，\w 等价于[0-9a-zA-Z_]。\s 的情况比较特殊，它既可以匹配 ASCII 编码中的空白字符，也可以匹配 Unicode 标准中的其他空白字符（比如中文的全角空格）。

```
//半角字符
/^\d$/u.test("1");      // => true
/^\w$/u.test("a");      // => true
/^\s$/u.test(" ");      // => true
//全角字符及中文字符
/^\d$/u.test("１");     // => false
/^\w$/u.test("我");     // => false
/^\s$/u.test("　");     // => true
```

ES2015 提供了 Unicode 匹配模式。按照文档说明，如果启用 Unicode 匹配模式，\d、\s、\w 的匹配规则不会变化，\D、\W、\S 的匹配规则会发生变化。经过笔者测试，发生变化的主要是一些特殊符号，而不是常见的字符，如果不会经常用到这些符号，姑且可以认为不会有变化。如果你确实希望使用 Unicode 匹配规则的\d、\w，可以用\p{Decimal_Number}替代\d，用[\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctuation}\p{Join_Control}]替代\w。

12.2.4 单词边界

JavaScript 的正则表达式中单词边界的规定与对\w 的规定是一样的，也就是说，\b 匹配成功只有一种情况：一边必须保证\w（等价于[0-9a-zA-Z_]）匹配成功，另一边必须保证\w 匹配不成功。因为 Unicode 模式并不会改变\w 的匹配，所以 Unicode 模式对单词边界没有影响。

```
//空字符串-英文字符
/a\b/u.test("a");       // => true
```

```
//英文字符-半角标点
/a\b/u.test("a,");      // => true
//英文字符-半角空格
/a\b/u.test("a ");      // => true
//英文字符-中文字符
/a\b/u.test("a我");     // => true
//英文字符-全角空格
/a\b/u.test("a　");     // => true
//中文字符-半角标点
/我\b/u.test("我,");    // => false
//中文字符-半角空格
/我\b/u.test("我 ");    // => false
//中文字符-中文字符
/我\b/u.test("我和");   // => false
//中文字符-全角空格
/我\b/u.test("我　");   // => false
```

12.2.5　行起始/结束位置

在 JavaScript 的正则表达式中，能够用来定位字符串或行起始位置的只有^（而没有\A）。默认情况下，^只能匹配“整个字符串的开始位置”；如果指定使用多行模式（Multiline Mode），^可以匹配字符串内部的文本行的开始位置。

```
/^1/u.test("1\n2");     // => true
/^2/u.test("1\n2");     // => false
/^2/mu.test("1\n2");    // => true
```

同样，能够定位字符串或行结束位置的只有$（而没有\z 和\Z）。默认情况下，$只能匹配整个字符串的结束位置。如果指定使用多行模式，$可以匹配字符串内部的文本行的结束位置。

```
/2$/u.test("1\n2");     // => true
/1$/u.test("1\n2");     // => false
/1$/mu.test("1\n2");    // => true
```

即便字符串末尾有换行符，$也只匹配字符串的结束位置，而不会匹配末尾换行符之前的位置。所以如果进行正则表达式验证，可以放心地在表达式中使用^和$。

```
/^\d{6}$/u.test("123456\n");     // => false
/^\d{6}$/u.test("123456");       // => true
```

12.2.6　环视

JavaScript 1.5 以后的版本（也就是说，除特别古老的 JavaScript 之外）都支持肯定顺序环视(?=…)和否定顺序环视(?!…)，环视中的表达式并没有任何限制。

```
/c(?=ab+)/u.test("cab");              // => true
/c(?=(ab+|cd))/u.test("cab");         // => true
/c(?=(ab+|cd))/u.test("cabbb");       // => true
```

传统的 JavaScript 的正则表达式中不支持肯定逆序环视`(?<=…)`和否定逆序环视`(?<!…)`，这确实非常不方便。好消息是，ES2017（TC39）已经提供了对逆序环视的支持，而且支持程度相当完美，逆序环视中能够使用的正则表达式没有任何限制。

```
/(?<=ab+)c/u.test("abc");             // => true
/(?<=(ab+|cd))c/u.test("abc");        // => true
/(?<=(ab+|cd))c/u.test("abbbc");      // => true
```

12.2.7 匹配模式

传统上，JavaScript 对匹配模式的支持非常有限，只支持不区分大小写模式（`i`）和多行模式（`m`），且不支持在正则表达式中用`(?modifier)`的记法设定模式，只能在末尾的反斜线之后写上模式对应的字母，所以模式必须对整个表达式生效。

```
/a/u.test("A");          // => false
/a/iu.test("A");         // => true
/1$/mu.test("1\n2");     // => true
/./u.test("\n");         // => false
```

好消息是，最新的 ES2017（TC39）中，已经提供了单行模式（`s`）。

```
/./su.test("\n");        // => true
```

在 JavaScript 中可以混用多个模式，具体做法是把模式对应的字母连续列出来，比如像下面这样指定不区分大小写模式和多行模式。

```
/^LINE2/imu.test("line1\nline2");     // => true
```

如果使用字符串形式生成 RegExp，则必须提供第二个参数来设定模式。

```
(new RegExp("A", "iu")).test("A");                    // => true
(new RegExp("line1$", "mu")).test("line1\nline2");    // => true
(new RegExp("^LINE2", "imu")).test("line1\nline2");   // => true
```

如果你确实需要匹配任意字符，又不能使用 ES2017（TC39），那么不妨使用`\s\S`或者`[\d\D]`，它们确实可以匹配包括换行符在内的任意字符。

```
//任何浏览器都成立
/line1[\s\S]/u.test("line1\rline2"); // => true
//对换行符也成立
/line1[\s\S]/u.test("line1\nline2"); // => true
```

JavaScript 中的另一种匹配模式是 `g`，它表示 Global（全局模式，对整个字符串执行），也就

是说，在找到第一次匹配之后，会持续进行正则表达式操作，直到最终找不到匹配为止。在提取文本和替换时，这个模式很有用，后文会详细介绍。

12.2.8　捕获分组的引用

在 JavaScript 中，如果要在正则表达式内部引用捕获分组，则应当使用*num* 记法，其中 *num* 为对应捕获分组的编号。

```
/^([a-z])\1$/u.test("ab");  // => false
/^([a-z])\1$/u.test("aa");  // => true
```

如果要在替换的 *replacement* 字符串中引用捕获分组，则应当使用$*num* 记法。

```
"2010-12-20".replace(/(\d{4})-(\d{2})-(\d{2})/u, "$2/$3/$1");
12/20/2010
```

如果要在 *replacement* 字符串中表示$字符，则必须转义为$$。

```
"the price is 12.99". replace(/\d+\.\d{0,2/u, "$$$0");
the price is $12.99
```

ES2017 新增了对命名分组的支持。如果使用了命名分组，具体写法是`(?<name>regex)`，其中 *name* 为捕获分组的名称。然后可以从结果中取到各个命名分组匹配的文本。

```
pattern = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
result = pattern.exec("2017-12-25");
// => result.groups.year === '2017'
// => result.groups.month === '12'
// => result.groups.day === '25'
```

如果要在正则表达式内部引用捕获分组，应当使用\k<*name*>记法，其中 *name* 为对应捕获分组的名字。

```
/^(?<char>[a-z])\k<char>$/u.test("aa");  // => true
/^(?<char>[a-z])\k<char>$/u.test("ab");  // => false
```

如果要在替换时引用捕获分组，应当使用$<*name*>记法，其中 *name* 为对应捕获分组的名字。

```
regex = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
replacement = "$<month>/$<day>/$<year>";
"2010-12-27".replace(regex, replacement);
// => "12/27/2010"
```

12.3 正则 API 简介

12.3.1 RegExp

JavaScript 中的正则表达式对象是 RegExp，生成它的方法主要有两种：使用/regex/分隔符表示法指定，或显式调用 RegExp()构造函数。

```
//简便形式，使用正则文字，不需要考虑字符串转义
var pattern = /\d+/u;
//指定了多行模式
var pattern = /\d+/mu;
```

显式使用 RegExp()构造函数有两种形式：直接以分隔符和正则文字表示，类似/regex/；或者以字符串作为参数，这时候就必须考虑字符串转义。如果使用前一种形式，必须在结束分隔符之后指定匹配模式；如果使用后一种形式，只能通过第二个参数指定匹配模式。

```
//常见形式，使用正则文字
var pattern = new RegExp(/\d+/u);
//以字符串形式给出，需要考虑字符串中的转义处理，\d 必须写成\\d
var pattern = new RegExp("\\d+");
//指定多行模式的两种方式
var pattern = new RegExp(/\d+/mu);
var pattern = new RegExp("\\d+", "mu");
```

第 6 章介绍过，一般情况下，编程语言中正则表达式对象和匹配对象是不同的，表达式对象不会包含与匹配有关的信息，但 JavaScript 中的 RegExp 是例外，它包含 5 个状态变量，如表 12-2 所示，其中的 lastIndex 尤其值得注意。

表 12-2 RegExp 中的变量

变量	说明
source	只读变量，保存文本形式的表达式
global	只读变量，标识正则表达式是否指定了全局模式（g）
ignoreCase	只读变量，标识正则表达式是否指定了不区分大小写模式（i）
multiline	只读变量，标识正则表达式是否指定了多行模式（m）
lastIndex	可写变量，如果使用了全局模式，这个变量保存的是在字符串中尝试下次匹配的偏移值，在 Regexp.test()和 Regexp.exec()中都会用到这个变量

如果使用 RegExp 的 *exec*()和 *test*()函数，并且设定了全局模式，正则表达式的匹配就会从 lastIndex 的位置开始，并且在每次匹配成功之后重新设定 lastIndex。这样，就可以在字符串中重复迭代，依次寻找各个匹配结果。但是，如果需要对不同的字符串调用同一个 RegExp 的 *exec*()或 *test*()方法，这个变量也有可能带来意料之外的匹配结果，所以应该记得更换字符串

时，显式将 RegExp 的 lastIndex 设定为 0。

12.3.1.1　RegExp.exec(string)

这个方法用来在字符串中寻找匹配，如果成功，则返回表示匹配信息的数组，否则返回 null。在默认情况下，它返回第一次匹配的结果。

```
var pattern = /\d{4}-\d{2}-\d{2}/u;
var str = "2010-12-20 2011-02-14";
if ((matchArray = pattern.exec(str))!= null) {
    console.log(matchArray[0] + " starts at " + matchArray.index);
}
2010-12-20 starts at 0
```

上一节说过，RegExp 对象中存在 lastIndex 变量，它指定下次开始尝试匹配的位置。但是如果没有指定全局模式，同时把判断语句从 if 改为 while，那么每次调用 RegExp.exec()时都会从字符串的起始位置开始尝试匹配，结果会造成无穷循环。

```
var pattern = /\d{4}-\d{2}-\d{2}/u;
var str = "2010-12-20 2011-02-14";
while ((matchArray = pattern.exec(str))!= null) {
    console.log(matchArray[0] + " starts at " + matchArray.index);
}
2010-12-20 starts at 0
2010-12-20 starts at 0
2010-12-20 starts at 0
……
```

指定全局模式 g 之后，RegExp 对象每次匹配成功，都会把匹配的结束位置更新到 lastIndex，下次调用时从 lastIndex 开始尝试匹配。

```
var pattern = /\d{4}-\d{2}-\d{2}/ug;
var str = "2010-12-20 2011-02-14";
while ((matchArray = pattern.exec(str))!= null) {
    //注意匹配结束位置是使用 pattern.lastIndex 获得的
console.log (matchArray[0] + " starts at " + matchArray.index + " and ends at "+
pattern.lastIndex);
}
2010-12-20 starts at 0 and ends at 10
2011-02-14 starts at 11 and ends at 21
```

你或许注意到了，上面的程序先把 RegExp 对象保存在变量 *pattern* 中，反复调用，而不是直接在 while 语句中写/\d{4}-\d{2}-\d{2}/ug.exec(str)。这是因为按照 ECMAScript 5 规范，除非复用 RegExp 对象，否则每次遇到/*regex*/或 new RegExp(*regex*)时，都会重新生成 RegExp 对象，这样就会造成死循环。

```
var str = "2010-12-20 2011-02-14";
while ((matchArray = /\d{4}-\d{2}-\d{2}/ug.exec(str))!= null) {
    console.log(matchArray[0] + " starts at " + matchArray.index);
}
2010-12-20 starts at 0
2010-12-20 starts at 0
2010-12-20 starts at 0
……
```

在 RegExp.exec(*string*)的匹配结果 matchArray 中，*index* 表示匹配文本在原文本中的开始位置，而 matchArray 数组中的各个元素，依次对应正则表达式中各个捕获分组匹配的文本。按照惯例，编号为 0 的分组对应整个正则表达式匹配的文本。在 JavaScript 中，只能获取各捕获分组匹配的文本，而不能获得各分组匹配文本的开始和结束位置。

```
var pattern = /(\d{4})-(\d{2})-(\d{2})/ug;
var str = "2010-12-20 2011-02-14";
while ((matchArray = pattern.exec(str))!= null) {
    console.log(matchArray[0] + " starts at " + matchArray.index + " and ends at "+
pattern.lastIndex);
    console.log(matchArray.slice(1).join(","));
}
2010-12-20 starts at 0 and ends at 10
2010,12,20
2011-02-14 starts at 11 and ends at 21
2011,02,14
```

关于变量 lastIndex 还有一点补充。ES6（ES2015）引入了新的“定点模式”（Sticky Mode），对应的模式修饰符是 y。它的意思是：在生成正则表达式的时候，即使还没有“看到”目标字符串，也把正则表达式对应的匹配位置固定下来。这样在真正匹配时，一旦指定的位置不能匹配成功，整个匹配即告失败，不会换其他位置重新尝试。而且匹配成功之后，lastIndex 也会对应变化。

```
pattern = /34/uy;
pattern.lastIndex = 3;
str = '012345678';
pattern.test(str));     // => true, 此时 lastIndex = 5
pattern.lastIndex = 1;
pattern.test(str);    // => false
```

你或许发现了，匹配模式 y 和匹配模式 g 的含义存在冲突。所以，如果同时指定了两种模式，则自动忽略模式 g。

12.3.1.2 RegExp.test(string)

此方法在本章开头已经介绍过，它用来测试正则表达式能否在字符串中找到匹配文本，返回布尔值。

```
/\d{4}-\d{2}-\d{2}/u.test("2010-12-20"));      // => true
```

在前端开发中，经常需要验证整个字符串是否能由正则表达式匹配。要完成这种功能，可以在表达式的两端分别加上匹配字符串起始和结束的^和$（注意：JavaScript 并不支持\A 和\z）。因为即便字符串末尾是换行符，$也会匹配真正的结束位置，所以可以放心使用。

```
/^\d{4}-\d{2}-\d{2}$/u.test("2010-12-20"));       // => true
/^\d{4}-\d{2}-\d{2}$/u.test("a2010-12-20"));      // => false
/^\d{4}-\d{2}-\d{2}$/u.test("2010-12-20\n"));     // => false
```

每个 RegExp 对象都包含状态变量 lastIndex。如果指定了全局模式，每次执行 RegExp.test()时，都会从字符串中的 lastIndex 偏移值开始尝试匹配，所以如果用同一个 *RegExp* 多次验证字符串，必须记得每次调用之后，将 lastIndex 设定为 0，否则就可能出错了。当然，更好的办法是调用 string.search(*RegExp*)，判断返回值是否为 0。

```
//保证使用同一个 RegExp 对象，且指定 g 模式
var pattern = /^\d{4}-\d{2}-\d{2}$/ug;
pattern.test("2010-12-20"));    // => true
pattern.test("2010-12-20"));    // => false
```

RegExp.test(*string*)还有一点奇妙之处：匹配成功之后，各捕获分组匹配的文本已经保存下来，用 RegExp.$1、RegExp.$2…就可以获得。不过，“理应”保存整个表达式所匹配文本的 RegExp.$0 并不存在。

```
if(/^(\d{4})-(\d{2})-(\d{2})$/u.test("2010-12-20")) {
    console.log(RegExp.$2 + "/" + RegExp.$3 + "/" + RegExp.$1);
}
12/20/2010
```

应当注意，如果当前匹配不成功，RegExp.$1 等变量仍然可能有值，只不过是最近一次成功调用 RegExp.test(*string*)时的值。

```
/^(\d{4})-(\d{2})-(\d{2})$/u.test("2010-12-20"); //匹配成功
/^(\d{4})-(\d{2})-(\d{2})$/u.test("abcd-ef-gh"); //匹配失败
console.log(RegExp.$2 + "/" + RegExp.$3 + "/" + RegExp.$1);
12/20/2010
```

12.3.2　String

JavaScript 中的一些正则操作（比如查找）既可以通过 RegExp 的方法实现，也可以通过 String 类的方法实现；另一些正则操作（比如切分和替换）是调用 String 类的方法实现的。

12.3.2.1　string.match(RegExp)

这个方法类似 RegExp.exec(*string*)，只是调换了 RegExp 和 string 的位置。两者的另一个区别是，无论是否指定全局模式，Regexp.exec()总是返回单次的匹配结果。对 string.match()

来说，如果指定了全局模式，则会返回一个字符串数组，其中包含各次成功匹配的文本，但不包含任何其他信息。

```
var pattern = /\d{4}-\d{2}-\d{2}/ug;
var str = "2010-12-20 2011-02-14";
var matchArray = str.match(pattern);
console.log(matchArray.join(","));
2010-12-20,2011-02-14
```

需要注意的是，如果指定了全局模式，即便正则表达式中包含捕获分组，在 string.match(*RegExp*)返回的数组中，也不会包含分组捕获的信息。

```
var pattern = /(\d{4})-(\d{2})-(\d{2})/ug;
var str = "2010-12-20 2011-02-14";
var matchArray = str.match(pattern);
console.log(matchArray.join(","));
2010-12-20,2011-02-14
```

如果没有指定全局模式，而正则表达式中包含捕获分组，则 string.match(*RegExp*)返回的数组中会包含分组捕获的信息。

```
var pattern = /(\d{4})-(\d{2})-(\d{2})/u;
var str = "2010-12-20 2011-02-14";
var matchArray = str.match(pattern);
console.log(matchArray.join(","));
2010-12-20,2011,12,20
```

12.3.2.2 string.search(RegExp)

这个方法用来寻找某个正则表达式在字符串中第一次匹配成功的位置，如果不成功，则返回-1。

```
"a1".search(/\d/u);     //  =>1
"ab".search(/\d/u);     //  =>-1
```

这个方法只能找到“第一次”匹配的位置，即便设定了全局模式，结果也不会有任何变化。所以，如果需要用一个表达式多次验证字符串，调用这个方法判断结果是否等于 0，是更好的办法。

```
var pattern = /^\d+$/ug;
"1234".search(pattern) == 0;     // => true
"5678".search(pattern) == 0;     // => true
```

相比之下，同样的 RegExp 对象，如果设定全局模式再调用 RegExp.test(*string*)，可能会因为 RegExp 中保存的 lastIndex 变量，得到错误的结果。

```
var pattern = /^\d+$/ug;
pattern.test("1234");  // => true
//这时候 lastIndex 的值是 4
pattern.test("5678");  // => false
```

12.3.2.3　string.replace(RegExp, replacement)

这个方法用来进行正则表达式替换，将 *RegExp* 能匹配的文本替换为 *replacement*。在默认情况下，它只进行一次替换，如果设定了全局模式，则所有能匹配的文本都会替换。

```
"2010-12-20 2011-02-14".replace(/\d{4}-\d{2}-\d{2}/u, "Date");
Date 2011-02-14

"2010-12-20 2011-02-14".replace(/\d{4}-\d{2}-\d{2}/ug, "Date");
Date Date
```

如果要在 *replacement* 字符串中引用分组，则可以使用*$num*。

```
"2010-12-20".replace(/(\d{4})-(\d{2})-(\d{2})/u, "$2/$3/$1");
12/20/2010
```

如果要在 *replacement* 字符串中表示$字符，则必须使用$$转义。

```
"the price is 12.99". replace(/\d+\.\d{0,2}/u, "$$$0");
the price is $12.99
```

在 *replacement* 字符串中，还有一些特殊变量可以使用，表 12-3 总结了常用的特殊变量。

表 12–3　JavaScript 的正则表达式中常用的特殊变量

特殊变量	说明
$num	表示对应捕获分组匹配的文本
$$	$字符
$`	正则表达式所匹配文本之前（左侧）的文本
$’	正则表达式所匹配文本之后（右侧）的文本

12.3.2.4　string.replace(RegExp, function)

这个方法可以算是上面方法的“变种”，只是把第二个参数从字符串变成了函数。这个函数的接收参数是一个字符串，返回参数也是一个字符串。这样一来，文本的处理就更加灵活了。下面的例子展示了如何将 t 开头的单词转换为大写。

```
console.log("one two three".replace(/\bt[a-zA-Z]+\b/ug,
function(m){return m.toUpperCase()}));
one TWO THREEE
```

12.3.2.5 string.split(RegExp)

这个方法使用一个正则表达式切分字符串，正则表达式是否使用了全局模式对结果没有影响。

```
var matchArray = "one two three".split(/\s+/u);
console.log(matchArray.join(","));
one,two,three
```

也可以设置第二个参数 *limit*，手工指定返回数组的数目。需要注意的是，在其他语言中，如果 *limit* 小于实际能切分的次数，返回数组的最后那个元素一般是“正则表达式第 *limit*-1 次匹配右侧的所有文本”，但在 JavaScript 中却不是这样，它仍然会进行尽可能多的切分，返回切分所得数组的第 *limit* 个元素。

```
//注意最后一个结果不是 two three
var matchArray = "one two three".split(/\s+/u, 2);
console.log(matchArray.join(", "));
one,two
```

表 12-4 总结了 *limit* 在各个取值区间对结果的影响（假定未指定 *limit* 时，最多可以切分 *n*-1 次，返回包含 *n* 个元素的数组）。

表 12–4 limit 各种取值的结果

取值	结果
`limit < 0`	等于未设定 limit 时，返回包含 *n* 个元素的数组
`0 <= limit <= n`	返回一个包含 *n* 个元素的数组，其中最后一个元素是切分 *n* 次时的第 limit 个元素（如果 limit 设定为 0，则返回空数组）
`limit > n`	等于未指定 limit 时

12.4 常用操作示例

12.4.1 验证

```
//必须在正则表达式首尾加上^和$
var regex = /^\d{4}-\d{2}-\d{2}$/u;
"2010-12-20".search(regex) == 0;     // => true

//如果使用 RegExp.test()，一定不能设定全局模式
var regex = /^\d{4}-\d{2}-\d{2}$/u;
regex.test("2010-12-20");            // => true
regex.test("2011-02-14");            // => true
```

12.4.2　提取

简单提取

```
var pattern = /\d{4}-\d{2}-\d{2}/ug;
var str = "2010-12-20 2011-02-14";
var matchArray = str.match(pattern);
console.log(matchArray.join(", "));
```

```
2010-12-20,2011-02-14
```

逐步提取

```
//如果要获取各捕获分组匹配的文本，只能使用这个方法，同时必须指定 g 模式
var pattern = /(\d{4})-(\d{2})-(\d{2})/ug;
var str = "2010-12-20 2011-02-14";
while ((matchArray = pattern.exec(str))!= null) {
    process.stdout.write("date: " + matchArray[0]);
    process.stdout.write(", year: " + matchArray[1]);
    process.stdout.write(", month: " + matchArray[2]);
    process.stdout.write(", day: " + matchArray[3]);
    process.stdout.write("\n");
}
```

```
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

逐步提取（使用命名分组）

```
//如果要获取各捕获分组匹配的文本，只能使用这个方法，同时必须指定 g 模式
var pattern = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/ug;
var str = "2010-12-20 2011-02-14";
while ((matchArray = pattern.exec(str))!= null) {
    process.stdout.write("date: " + matchArray[0]);
    process.stdout.write(", year: " + result.groups.year);
    process.stdout.write(", month: " + result.groups.month);
    process.stdout.write(", day: " + result.groups.day);
    process.stdout.write("\n");
}
```

```
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

12.4.3 替换

简单替换

```
var regex = /(\d{4})-(\d{2})-(\d{2})/u;
var replacement = "$2/$3/$1";
console.log("2010-12-20".replace(regex, replacement));
12/20/2010
```

简单替换（使用命名分组）

```
var regex = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/u;
var replacement = "$<month>/$<day>/$<year>";
console.log("2010-12-20".replace(regex, replacement));
12/20/2010
```

在 replacement 中使用$

```
var regex = /\d+\.\d{0,2}/u;
var replacement = "$$$0";
console.log("the price is 12.99". replace(regex, replacement));
the price is $12.99
```

12.4.4 切分

```
var regex = /\s+/u;
var matchArray = "one\ntwo\t three".split(regex);
for(var i = 0; i < matchArray.length; i++) {
    console.log(matchArray.join(","));
}
one,two,three
```

12.5 关于 ActionScript

ActionScript 是用于 Adobe Flash 和 Adobe AIR 运行时环境的编程语言，它同样遵循 ECMAScript 标准（只不过它遵循的是 ECMAScript 4，ES4 是一个非常激进的版本，变成了强类型语言，所以没有被浏览器和开发者认同，JavaScript 其实也没有基于 ES4 的版本）。ActionScript 中的正则表达式操作与 JavaScript 非常相似，所以在这里简要介绍 ActionScript 中的正则表达式。

12.5.1 RegExp

与 JavaScript 一样，ActionScript 中的正则对象同样是 `RegExp`，只是生成方式有细微区别，在 ActionScript 中，必须显式指定所生成对象的类型为 `RegExp`。

```
//通过正则文字生成
var pattern1:RegExp = /bob/;
//通过正则文字生成，同时指定不区分大小写模式
var pattern1:RegExp = /bob/i;
//通过字符串生成
var pattern2:RegExp = new RegExp("bob");
//通过字符串生成，同时指定不区分大小写模式
var pattern2:RegExp = new RegExp("bob", "i");
```

12.5.2　匹配规则

ActionScript 只支持 ASCII 匹配规则，所以`\w`、`\s`、`\d`都无法匹配 ASCII 编码之外的字符。

```
//半角字符
/^\d$/.test("1");  // => True
/^\w$/.test("a");  // => True
/^\s$/.test(" ");  // => True
//全角字符及中文字符
/^\d$/.test("１");// => False
/^\w$/.test("我");// => False
/^\s$/.test("　");// => False
```

12.5.3　匹配模式

相比 JavaScript，ActionScript 对匹配模式的支持要好很多，它一共支持 5 种匹配模式，如表 12-5 所示。

表 12–5　ActionScript 中的匹配模式

模式修饰符	说明
g	全局模式，意义与 JavaScript 的相同，用于提取多个结果
i	不区分大小写模式
s	单行模式
m	多行模式
x	注释模式

同样，ActionScript 中的 `RegExp` 对象也会保留 `lastIndex` 信息。所以，如果你希望对某个字符串重复应用同一个正则表达式，依次查找，请记得指定全局模式。

12.5.4　正则 API

ActionScript 中的正则 API 与 JavaScript 中的几乎完全一致，为避免重复，这里不做重复说明。

第 13 章　PHP

PHP内建了三套各自独立的正则引擎，分别是preg、ereg、mb_ereg。不过日常使用最多的是preg（preg表示PCRE Regex，PCRE表示Perl Compatible Regular Expression，常见的正则表达式写法都是兼容Perl的），本章的讲解也以preg为准。[1]

13.1　预备知识

PHP 的正则表达式是以字符串文字指定的，不过又不同于普通的字符串文字，如果用一段字符串文字表示正则表达式，文字的首位必须加上分隔符（delimeter），通常使用的分隔符是斜线/，在 4.0.4 以上的版本中，也可以采用()、{}、<>、[]这些成对的符号。不过()、{}、[]等符号都是正则表达式中常用的元字符，如果用作分隔符，用作元字符时就也必须转义，非常烦琐，而/在正则表达式中没有特殊含义，所以一般推荐用/做分隔符，本章中也是如此。

```
$regex = '/regex/';     //这是正确的写法
$regex = '{regex}';     //这也是正确的写法
$regex = 'regex';       //这是常犯的错误，正则表达式没错，但忘了加分隔符
```

你可能注意到了，上面的字符串文字是用单引号标注的。在 PHP 中，字符串既可以用双引号标注，也可以用单引号标注。两者的主要区别在于，双引号字符串可以插值，而单引号字符串不能；另外，双引号字符串会处理字符串转义，而单引号字符串不会。

```
$regex = "[a-z]";
echo "double-quoted\t$regex";
echo "<br />";
echo 'single-quoted\t$regex';
```

```
double-quoted [a-z]
single-quoted\t$regex
```

---

[1] 如果不确定计算机中是否有 PCRE，可以按照下面的方法测试，这里附上我的测试结果。

yusheng@my_ubuntu:~$ pcretest -C

PCRE version 7.8 2008-09-05 ……

如果不需要插值，使用双引号字符串的处理效率就不如单引号字符串，再加上双引号字符串用作正则表达式时还需要处理字符串转义，所以本章表示正则表达式时，都使用单引号字符串直接写出正则文字（除非是 PHP 7 中新出现的`\u{xxxx}`转义序列，它必须使用双引号字符串，否则编译不通过），而字符串文字则使用双引号字符串，方便识别`\r`、`\n` 之类常见的转义序列。

下面介绍正则表达式匹配的函数 `preg_match($pattern, $subject)`。它接收两个参数，其中 *pattern* 是正则表达式，*subject* 是需要匹配的文本；返回的是匹配发生的次数——如果能找到匹配则立刻停止，返回 1，否则返回 0；如果匹配过程中发生错误，则返回 `false`。需要注意的是，它并不检查整个 *subject* 能否由 *pattern* 匹配。

```
preg_match('/\d/', "1");    // => True
preg_match('/\d/', "a1b");  // => False
```

注：因为 PHP 可以把 int 类型作为布尔判断的条件，0 等价于 false，其他值等价于 true。在下面的代码里，为了形象起见，如果只关心 `preg_match()` 的返回值是否为 0，则使用 true 和 false 来标识。

在正则表达式两端添加`\A`和`\z`，就可以判断整个 *subject* 能否由 *pattern* 匹配。

```
preg_match('/\A\d\z/', "1");    // => True
preg_match('/\A\d\z/', "a1b");  // => False
```

在有些 PHP 代码中，使用双引号字符串来表示正则表达式，某些时候，情况看起来有点奇怪。

```
preg_match("/\d/", "1");    // => True
preg_match("/\\d/", "1");   // => True
```

尽管 PHP 的正则表达式用字符串文字给出，但它与常见的字符串不完全一样，主要是转义规则有所区别——如果某个转义序列可以由字符串识别，则对其进行转义处理；否则，将整个转义序列“原封不动”地保存下来。在处理正则表达式时，“原封不动”保存下来的文字，会作为正则文字，只进行正则转义处理（详见第 95 页）。

这种现象可能很难理解：`/\d/`和`/\\d/`看起来不同，却表示同样的正则表达式。所以，我推荐的做法是（理由见第 6 章）：正则表达式尽量使用单引号字符串，比如`'/\d/'`或`'/\A/'`；如果必须使用双引号字符串，尽量写为`"/\\d/"`或`"/\\A/"`，因为在大多数语言中都可以这么写。

关于 mb_string

用过 PHP 的人大都知道 PHP 中的 mb_string，它用来处理多字节字符构成的字符串（multibyte-string），不会将多字节字符拆散为多个单字节字符。比如用 `mb_strpos()`来计算偏移值，得到偏移值的单位就是字符，而不是字节。

```
strpos("位置", "置", 0);              //3
mb_strpos("位置", "置", 0, "utf-8");  //1
```

相应的，mb_string 中也提供了正则表达式函数，比如 `mb_ereg_match()`、`mb_ereg_replace()`、

mb_regex_encoding()等，其功能类似对应的 preg 函数，具体情况请参考 http://www.php.net/manual/en/ref.mbstring.php。

按照文档说明，在设定 mb_regex_encoding()之后，可以正确处理非 Unicode 编码的正则表达式，但实际使用中往往容易出现各种奇怪的问题（例如上面提到的[正则]的错误匹配，即便设定了 GBK 编码，也会出现），所以一般不推荐这样使用正则表达式。

PHP 7 与 Unicode

历史上，PHP 一直没有很好的办法来处理 Unicode。比如我们知道“正”字的 Unicode 码值是 U+6B63，但是没法在代码里通过\u6B63 这样的方式来指定它。PHP 虽然提供了\x 的转义序列，但只支持两位十六进制数，所以\u6B63 会被解释为 3 个字符：k（码值为 6B 的字符）、6、3。

```
echo("\x6b63");              //k63
```

这个问题在 PHP 7 中得到了很好的解决，新提供的\u{*xxxx*}转义序列完美支持了通过码值指定 Unicode 字符的方式。而且转义序列不限于 4 位，也就是说，超出 BMP 之外的其他 Unicode 字符（包括 Emoji）也可以通过这种方式来指定。所以如果使用 PHP 7，最好统一使用\u{*xxxx*}转义序列。同时要注意的是，一旦使用了这种转义序列，就必须使用双引号字符串。

```
echo("\u{6b63}");            //正
echo("\u{1F60A}");           //☺
```

13.2 正则功能详解

13.2.1 列表

表 13-1 列出了 PHP 中的正则功能。

表 13–1 PHP 中的正则功能

功能	记法	说明
字符组 ☞2	[…] [^…]	支持 Unicode
POSIX 字符组 ☞15	[:digit:]	只匹配 ASCII 字符
Unicode 属性 ☞127	\p{…}	支持
字符组简记法 ☞120	\d \D \w \W \s \S	采用 ASCII 匹配规则
行起始位置 ☞62	^ \A	支持
行结束位置 ☞62	$ \z \Z	支持
单词边界 ☞123	\b \B	采用 ASCII 匹配规则

（续表）

功能	记法	说明
顺序环视 ☞69	`(?=…)` `(?!…)`	支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	不支持无法确定长度范围的表达式
匹配模式 ☞83	`i m s x`	支持
模式作用范围 ☞91	`(?modifier)` `(?-modifier)`	支持
纯文本模式 ☞101	`\Q…\E`	支持
捕获分组及引用☞44	`(…)` `\num` `$num`	支持
命名分组 ☞53	`(?<name>…)` `\k<name>`	支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

13.2.2　字符组

PHP 中的正则引擎可以正确识别 ASCII 编码，但无法正确识别 GBK 编码，因此无法避免 GBK 编码环境中的“错误匹配”问题（☞116）。

```
preg_match('/[正则]/', "遭"); // => True
preg_match('/[正则]/', "正"); // => True
```

解决的办法是采用 Unicode 编码环境（以 Unicode 编码保存源文件），并在正则表达式末尾的分隔符之后加上模式修饰符 u，它指定 Unicode 模式，也就是按 UTF-8 编码解释正则表达式。

```
preg_match('/[正则]/u', "遭"); // => False
```

只有在指定 Unicode 模式之后，点号才能准确匹配多字节字符。

```
# GBK 编码环境
preg_match('/\A.\z/u', "遭"); // => False
# Unicode 编码环境
preg_match('/\A.\z/u', "遭"); // => True
```

PHP 的正则表达式中（注意，不是字符串中）支持使用`\x{`*hex*`}`通过码值表示字符，其中的 *hex* 是 4 位十六进制数，所以可以用`[\x{4E00}-\x{9FFF}]`来匹配汉字字符，但是此时一定不能忘记 u 修饰符。

```
preg_match('/[\x{4E00}-\x{9FFF}]/u', "我"); // => True
preg_match('/[\x{4E00}-\x{9FFF}]/u', "a");  // => False
preg_match('/[\x{4E00}-\x{9FFF}]/', "我");  //错误
```

如果使用的是 PHP 7，也可以直接使用新提供的\u{*hex*}，此时可以不使用 u 修饰符，但必须使用双引号字符串。

```
preg_match("/[\u{4E00}-\u{9FFF}]/", "我");    // => True
preg_match("/[\u{4E00}-\u{9FFF}]/", "a");     // => False
preg_match('/[\u{4E00}-\u{9FFF}]/', "我");    //编译错误
```

在 PHP 中也可以直接使用 POSIX 字符组，所有的 POSIX 字符组都只能匹配 ASCII 字符，即便指定了 Unicode 模式也是这样。

```
preg_match('/[[:digit:]]/', "0");    // => True
preg_match('/[[:xdigit:]]/', "f");   // => True
```

13.2.3 Unicode 属性

从PHP 5.1.0 开始，PHP支持使用Unicode属性，方法是\p{N}和\P{N}，其中N为Unicode Property（☞130），而且此时一定不能忘记u修饰符。[1]

\p{N}可以匹配“任何一个具备指定 Unicode Property 的字符”，而\P{N}匹配“任何一个不具备指定 Unicode Property 的字符”，以表示大写字母的 Lu 为例。

```
preg_match('/\p{Lu}/u', "A"); // => True
preg_match('/\p{Lu}/u', "a"); // => False

preg_match('/\P{Lu}/u', "A"); // => False
preg_match('/\P{Lu}/u', "a"); // => True
```

\P{N}也可以等价表示为\p{^N}：

```
preg_match('/\p{^Lu}/u', "A");  // => False
preg_match('/\p{^Lu}/u', "a");  // => True
```

因为 PHP 中可以使用 Unicode Script（☞129），所以可以用\p{Han}来匹配中文字符。

```
preg_match('/\p{Han}/u', "我"); // => True
```

13.2.4 字符组简记法

在 PHP 中，\d、\D、\w、\W、\s、\S 都采用字符组简记法的 ASCII 匹配规则，即便指定了 Unicode 模式，也是如此。

```
//半角字符
preg_match('/\d/', "0"); // => True
preg_match('/\w/', "a"); // => True
preg_match('/\s/', " "); // => True
```

[1] 请参考 http://www.php.net/manual/en/regexp.reference.unicode.php。

```
//全角字符
preg_match('/\d/u', "０"); // => False
preg_match('/\w/u', "ａ"); // => False
preg_match('/\s/u', "　"); // => False
```

13.2.5　单词边界

在PHP中，\w采用ASCII匹配规则，所以\b能匹配位置的某一侧必须出现[0-9a-zA-Z_]，另一侧不能出现[0-9a-zA-Z_]。

```
//空字符串-英文字符
preg_match('/a\b/', "a");       // => True
//英文字符-半角标点
preg_match('/a\b/', "a,");      // => True
//英文字符-半角空格
preg_match('/a\b/', "a ");      // => True
//英文字符-中文字符
preg_match('/a\b/', "a我");     // => True
//英文字符-全角空格
preg_match('/a\b/', "a　");     // => True
//中文字符-半角标点
preg_match('/我\b/', "我,");    // => False
//中文字符-半角空格
preg_match('/我\b/', "我 ");    // => False
//中文字符-中文字符
preg_match('/我\b/', "我和");   // => False
//中文字符-全角空格
preg_match('/我\b/', "我　");   // => False
```

前面讲过，PHP处理双引号字符串时，会将字符串文字中无法识别的转义序列保留下来，所以"/\d/"和"/\\d/"是没有差别的。值得注意的是，"/\b/"和"/\\b/"也是没有差别的，因为PHP的字符串转义并不能识别转义序列\b。

13.2.6　行起始/结束位置

在PHP的正则表达式中，^匹配的是“行开始的位置”，在默认情况下，它只能匹配“整个字符串的开始位置”；如果指定使用多行模式（Multiline Mode），^可以匹配字符串内部的文本行的开始位置；而\A无论在什么情况下都匹配“整个字符串的开始位置”。

```
preg_match('/^1/', "1\n2\n");   // => True
preg_match('/^2/', "1\n2\n");   // => False
preg_match('/^2/m', "1\n2\n");  // => True

preg_match('/\A1/', "1\n2\n");  // => True
preg_match('/\A2/', "1\n2\n");  // => False
```

```
preg_match('/\A2/m', "1\n2\n"); // => False
```

在默认情况下，PHP 中的$、\Z、\z 匹配的是整个字符串的结束位置（如果结束位置有换行符，则$和\Z匹配这个换行符之前的位置）；使用多行模式会影响到$的匹配，这时候它可以匹配文本内部行的结束位置；如果要进行严格准确的验证，则应当使用\z。

```
preg_match('/1$/', "1\n");      // => True
preg_match('/1$/', "1\n2");     // => False
preg_match('/1$/m', "1\n2");    // => True

preg_match('/1\Z/', "1\n");     // => True
preg_match('/1\Z/', "1\n2");    // => False
preg_match('/1\Z/m', "1\n2");   // => False

preg_match('/1\z/', "1");       // => True
preg_match('/1\z/', "1\n");     // => False
preg_match('/1\z/m', "1\n");    // => False
```

13.2.7 环视

PHP 中的肯定顺序环视(?=…)和否定顺序环视(?!…)内可以嵌套任意形式的子表达式。

```
preg_match('/c(?=ab+)/', "cab");          // => True
preg_match('/c(?=(ab+|cd))/', "ccd");     // => True
preg_match('/c(?=(ab+|cd))/', "cabbbb"); // => True
```

PHP 的逆序环视中能使用的表达式，必须匹配固定长度的文本，否则会报错。如果表达式中出现了多选结构，而各个分支能匹配的文本长度不一致，那么这个多选结构必须位于表达式的顶层，而且只能出现竖线|，类似bull|donkey，而不能出现括号，比如(bull|donkey)。

```
preg_match('/ab(?<=ab)/', "ab");     // => True
preg_match('/cd(?<=ab|cd)/', "cd");  // => True

//长度不固定，报错
preg_match('/cd(?<=(dogs?|cats?))/', "cd");
//多选结构各分支匹配文本长度不一致，且两端出现了括号，报错
preg_match('/cd(?<=(bull|donkey))/', "cd");
```

13.2.8 匹配模式

表 13-2 列出了 PHP 中常用的匹配模式。

表 13–2 PHP 中常用的匹配模式

匹配模式	修饰符	解释
不区分大小写模式	i	不区分 ASCII 字符的大小写

（续表）

匹配模式	修饰符	解释
注释模式	x	允许正则表达式中出现注释，表达式中的空白字符，以及#开始到行末的文本，都视为注释
多行模式	m	^和$不仅仅匹配字符串的起始/结束位置，还可以匹配字符串内部文本行的起始/结束位置
单行模式	s	允许点号.匹配任何字符，包括换行符
Unicode 模式	u	UTF-8 模式，不会把正则表达式中的多字节字符拆散为字节，而是将其视为单个字符

PHP 中指定匹配模式的方式有两种：在结尾分隔符之后加上模式对应的字母，或者在正则表达式的开头用`(?modifier)`的方式表示。注意：在 PHP 中没有通过预定义常量指定模式的方法。

```
preg_match('/a/', "A");     // => False
preg_match('/a/i', "A");    // => True
preg_match('/(?i)a/', "A"); // => True
```

其中 Unicode 模式有些特殊，它只能在末尾分隔符之后使用，不能使用`(?modifier)`的记法。

```
preg_match('/\A.\z/u', "字");        // => True
preg_match('/(?u)\A.\z/', "字");     //编译错误
```

PHP 也支持用`(?-modifier)`结束某个模式的作用范围。

```
preg_match('/a(?i)b(?-i)b/', "abb");     // => True
preg_match('/a(?i)b(?-i)b/', "aBb");     // => True
preg_match('/a(?i)b(?-i)b/', "aBB");     // => False
```

13.2.9　纯文本模式

在`\Q`和`\E`之内，所有的元字符和特殊结构都失去特殊意义，只能匹配它们对应的字符本身。

```
preg_match('/\Q.\E/', "A");     // => False
preg_match('/\Q.\E/', ".");     // => True
```

13.2.10　捕获分组的引用

在 PHP 中，如果要在正则表达式内部引用捕获分组，则应当使用*\num* 记法，其中 *num* 为对应捕获分组的编号。

```
preg_match('/([a-z])\1/', "ab");     // => False
preg_match('/([a-z])\1/', "aa");     // => True
```

如果要在替换时引用捕获分组，则应当使用*$num* 记法，不过更好的办法是使用*${num}*记法，这样可以避免二义性（☞50）。

```
preg_replace('/(\d{4})-(\d{2})-(\d{2})/', "$2/$3/$1", "2010-12-20");
12/20/2010

preg_replace('/(\d{4})-(\d{2})-(\d{2})/', "${2}/${3}/${1}", "2010-12-20");
12/20/2010
```

如果希望在 *replacement* 字符串中表示$字符，则必须使用\$转义。

```
preg_replace('/\d+\.\d{0,2}/', '\$${0}', "the price is 12.99");
the price is $12.99
```

PHP 支持命名分组，分组的记法是(?P<*name*>…)，在表达式中引用的记法为(?P=*name*)（在 5.2.2 版本以后可以使用\k<*name*>或者\k'*name*'，在 5.2.4 版本之后可以使用\k{*name*}和\g{*name*}），在替换中，却只能通过数字编号来引用。

```
preg_match('/(?P<char>[a-z])(?P=char)/', "ab"); // => False
preg_match('/(?P<char>[a-z])(?P=char)/', "aa"); // => True
```

需要指出的是，如果进行正则表达式替换，*replacement* 中不能使用命名分组，只能使用数字编号分组。

13.3 正则 API 简介

13.3.1 PREG 常量说明

preg 系列函数有可能用到预定义的常量，所以先介绍这些常量。

13.3.1.1 PREG_PATTERN_ORDER

仅用于 preg_match_all()，在匹配结果的 matches 数组中，matches[0]保存每次匹配成功时捕获的全部文本（也就是编号为 0 的捕获分组匹配的文本）；matches[1]保存每次匹配成功时编号为 1 的捕获分组匹配的文本；依此类推。

13.3.1.2 PREG_SET_ORDER

仅用于 preg_match_all()，在匹配结果的 matches 数组中，matches[0]保存每一次匹配成功时，整个表达式以及各捕获分组匹配的文本；matches[1]保存第 2 次匹配成功时，整个表达式以及各捕获分组匹配的文本；依此类推。

13.3.1.3 PREG_SPLIT_OFFSET_CAPTURE

在 PHP 4.3.0 以上版本中可用，仅用于 preg_split()中。如果设定了这个常量，每次匹配之后，会返回后面的字符串的偏移值。请注意，返回值是一个数组，其中每个元素都是一个数组，包含偏移值为 0 的匹配的字符串，以及它在文本中的偏移值，从 1 开始。

13.3.1.4　PREG_OFFSET_CAPTURE

在 PHP 4.3.0 以上版本中可用，其功能类似 PREG_SPLIT_OFFSET_CAPTURE，但它用于 preg_match()和 preg_matchall()，如果此时指定 PREG_SPLIT_OFFSET_CAPTURE，则会报错。

13.3.1.5　PREG_SPLIT_NO_EMPTY

在 preg_split()中使用它，返回结果中不会包含空字符串。

13.3.1.6　PREG_SPLIT_DELIMI_CAPTURE

在 PHP 4.0.5 以上版本中可用，在 preg_slipt()中使用它，则会捕获正则表达式中的捕获括号所匹配的文本。

下面几个常量都是 preg_last_error()返回的。

13.3.1.7　PREG_NO_ERROR

在 PHP 5.2.0 以上版本中可用，如果没有错误，则返回它。

13.3.1.8　PREG_INTERNAL_ERROR

在 PHP 5.2.0 以上版本中可用，如果 PCRE 发生了内部错误，则返回它。

13.3.1.9　PREG_BACKTRACK_LIMIT_ERROR

在 PHP 5.2.1 以上版本中可用，如果尝试匹配时超过了回溯限制，则返回它。

13.3.1.10　PREG_RECURSION_LIMIT_ERROR

在 PHP 5.2.0 以上版本中可用，如果匹配时超过了递归限制，则返回它。

13.3.1.11　PREG_BAD_UTF8_ERROR

在 PHP 5.2.0 以上版本中可用，如果在 UTF-8 模式下正则表达式中出现了不合法的 UTF-8 数据，则返回它。

13.3.1.12　PREG_BAD_UTF8_OFFSET_ERROR

在 PHP 5.3.0 以上版本中可用，如果在 UTF-8 模式下起始偏移值对应的不是有效的 UTF-8 码值，则返回它。

13.3.1.13　PCRE_VERSION

在 PHP 5.2.4 以上版本中可用，返回 PCRE 版本号和发布日期。

13.3.2 preg_quote

```
string preg_quote ( string $str [, string $delimiter = NULL ] )
```

这个函数用来“包装”字符串 *str*。如果需要把这个字符串用作正则表达式，又要保证其中元字符都没有特殊含义（比如处理用户输入的内容），则可以使用它。

```
echo preg_quote('^.$');
\^\.\$
```

虽然在 PHP 中使用正则表达式时必须在首尾加上分隔符（delimiter），但分隔符可以使用多种字符。所以，preg_quote()中的 *delimiter* 是可选参数；尽管一般用斜线/做分隔符，但在默认情况下，*str* 中的斜线并不会转义，除非显式指定使用/作为分隔符。

```
echo preg_quote('/');
/

echo preg_quote('/', '/');
\/
```

如果要由某个字符串生成正则表达式，同时保证其中不包含任何元字符，可以用下面的办法（别忘了加上分隔符）。

```
$delimiter = '/';
$regex = $delimiter.preg_quote('.*/', $delimiter).$delimiter;
preg_match($regex, '.*/');  // => True
```

也可以直接在字符串两端加上\Q 和\E。

```
$delimiter = '/';
$regex = $delimiter.'\Q'.origin.'\E'.$delimiter;
preg_match($regex, '.*/');  // => True
```

13.3.3 preg_ grep

```
array preg_grep ( string $pattern, array $input [, int $flags = 0])
```

这个函数用来“筛选”数组，*input*数组中所有能由正则表达式*pattern*匹配的字符串会被筛选出来，重新组成一个数组，作为结果返回。[1]

```
preg_grep('/^\d$/', array("1", "a", "2", "b"));
Array{ [0]=>"1", [2]=>"2" }
```

如果可选参数 *flags* 设定为常量 PREG_GREP_INVERT，则进行“反向筛选”，也就是说，*input*

[1] 在本章中，为方便展示返回数组的内容，数组使用 var_dump()函数展开的形式来表现。

中所有 *pattern* 不能匹配的字符串都被筛选出来，重新组成一个数组，作为结果返回。

```
preg_grep('/^\d$/', array("1", "a", "2", "b"), PREG_GREP_INVERT);
Array{ [0]=>"a", [3]=>"b" }
```

flags 参数在 PHP 4.2.0 以后才加入。

13.3.4　preg_match

```
int preg_match ( string $pattern, string $subject [, array &$matches [, int $flags = 0 [, int $offset = 0 ]]] )
```

这是很常用的函数，它在 *subject* 中查找 *pattern* 的匹配，返回匹配的次数。但是，它会在第一次匹配成功之后停下来，直接返回 1；如果始终不能匹配成功，则返回 0；如果发生错误，则返回 false。

```
preg_match('/\d/', "0123");     // => True
preg_match('/\d/', "abcd");     // => False
```

根据 preg_match()的返回值，可以方便地进行数据验证，但是注意在正则表达式的首尾加上\A和\z，确保它验证的是整个字符串。下面以验证 6 到 8 位数字的字符串为例。

```
preg_match('/\A\d{6,8}\z/', "012345");        // => True
preg_match('/\A\d{6,8}\z/', "0123456789");    // => False
preg_match('/\A\d{6,8}\z/', "a012345");       // => False
```

可选参数 *matches* 用来保存匹配结果，如果匹配成功，它包含一个元素，即首次匹配的结果，否则数组为空。

```
$matches = array();
preg_match('/\d/', "0123", $matches);
$matches: Array{ [0]=>"0" }

$matches = array();
preg_match('/\d/', "abcd", $matches);
$matches: Array{ }
```

如果 *pattern* 中出现了捕获分组，则在 *matches* 数组中与捕获分组编号对应的元素保存该捕获分组匹配的文本。

```
$matches = array();
preg_match('/(\d)(\d)/', "0123", $matches);
$matches: Array{ [0]=>"01", [1]=>"0", [2]=>"1" }
```

可选参数 *flags* 控制匹配结果中是否保存匹配发生的位置，如果设置为常量 PREG_OFFSET_CAPTURE，则保存匹配发生的位置，否则可以设置为 0。

```
$matches = array();
preg_match('/\d/', "0123", $matches, PREG_OFFSET_CAPTURE);
$matches: Array{ [0]=> Array{[0]=>"0", [1]=>0} }

$matches = array();
preg_match('/\d/', "0123", $matches, 0);
$matches: Array{ [0]=>"0" }
```

应该注意的是，位置的单位是字节，而不是字符。

```
$matches = array();
preg_match('/\d/', "字0123", $matches, PREG_OFFSET_CAPTURE);
$matches: Array{ [0]=> Array{[0]=>"0", [1]=>4} }
```

可选参数 *offset* 控制匹配从 *subject* 的哪个位置开始。

```
$matches = array();
preg_match('/\d/', "0123", $matches, PREG_OFFSET_CAPTURE, 2);
$matches: Array{ [0]=> Array{[0]=>"2", [1]=>2} }
```

注意：这里位置的单位也是字节，而不是字符。

```
$matches = array();
preg_match('/\d/', "0和123", $matches, PREG_OFFSET_CAPTURE, 4);
$matches: Array{ [0]=> Array{[0]=>"1", [1]=>4} }
```

flags 参数和 PREG_OFFSET_CAPTURE 都是 PHP 4.3.0 之后加入的，*offset* 参数则是 PHP 4.3.3 之后加入的。

13.3.5 preg_match_all

```
int preg_match_all (string $pattern, string $subject, array &$matches[, int
$flags = PREG_PATTERN_ORDER[, int $offset = 0 ]])
```

这个函数与 preg_match()相似，不同的是，它可以一次性找出正则表达式 *pattern* 在字符串 *subject* 中所有的匹配，返回匹配的次数；如果始终无法匹配，则返回 0；如果出错，则返回 false。在 preg_match_all()中，同样也可以用 *matches* 数组来保存匹配结果。

```
$matches = array();
preg_match_all('/\d{4}-\d{2}-\d{2}/', "2010-12-20 2011-02-14", $matches);
$matches: Array{ [0]=>"2010-12-20", [1]=>"2011-02-14" }
```

如果 *pattern* 中出现了捕获型括号，则 *matches* 的每个元素都是数组，它保存对应编号的捕获分组在历次匹配中匹配到的文本。

```
$matches = array();
preg_match_all('/(\d{4})-(\d{2})-(\d{2})/', "2010-12-20 2011-02-14", $matches);
```

```
$matches: Array {
   [0]=> array(2)
        { [0]=>"2010-12-20", [1]=>"2011-02-14" },
   [1]=> array(2)
        { [0]=>"2010", [1]=>"2011" },
   [2]=> array(2)
        { [0]=>"12", [1]=>"02" },
   [3]=> array(2)
        { [0]=>"20", [1]=>"14" }
}
```

可以看到，*matches* 数组的第 0 个元素对应两次匹配时整个 *pattern* 匹配的文本（也就是第 0 个捕获分组匹配的文本），第 1 个元素对应两次匹配时 *pattern* 中第 1 个捕获分组匹配的文本，第 2 个元素对应两次匹配时 *pattern* 中第 2 个捕获分组匹配的文本。这是默认的方式，*flags* 显式设定为 PREG_PATTERN_ORDER 也是这样。

如果你在其他语言中使用过正则表达式，可能会觉得这样非常别扭，因为一般的逻辑中，匹配结果集中的每个元素对应到“每次匹配”的结果，而不是“每个分组匹配”的结果。如果希望采用更符合惯例的逻辑，需要将 *flags* 设定为 PREG_SET_ORDER。

```
$matches = array();
preg_match_all('/(\d{4})-(\d{2})-(\d{2})/', "2010-12-20 2011-02-14", $matches,
PREG_SET_ORDER);
```

```
$matches: Array {
   [0]=> array(4)
        { [0]=>"2010-12-20", [1]=>"2010", [2]=>"12", [3]=>"20" },
   [1]=> array(4)
        { [0]=>"2011-02-14", [1]=>"2011", [2]=>"02", [3]=>"14" }
}
```

```
$matches = array();
preg_match_all('/(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})/', "2010-12-20 2011-02-14",
$matches, PREG_SET_ORDER);
```

```
$matches: Array {
   [0]=> Array {
     [0]=>"2010-12-20", ["year"]=>"2010", [1]=>"2010",
     ["month"]=> "12", [2]=>"12", ["day"]=>"20", [3]=>"20" },
   [1]=> Array {
     [0]=>"2011-02-14", ["year"]=>"2011", [1]=>"2011",
     ["month"]=>"02", [2]=>"02", ["day"]=>"14", [3]=>"14" }
}
```

flags 参数也可以设定为 PREG_OFFSET_CAPTURE，这样 *matches* 的结果相比 PREG_PATTERN_ORDER，就多出了匹配发生的位置信息（同样，单位是字节）。

```
$matches = array();
preg_match_all('/(\d{4})-(\d{2})-(\d{2})/', "2010-12-20 2011-02-14", $matches, PREG_ OFFSET_CAPTURE);

$matches: Array {
   [0]=> array(2)
       { [0]=>"2010-12-20", [1]=>"2011-02-14" },
   [1]=> array(2)
       { [0]=>"2010", [1]=>"2011" },
   [2]=> array(2)
       { [0]=>"12", [1]=>"02" },
   [3]=> array(2)
       { [0]=>"20", [1]=>"14" }
}
```

PREG_OFFSET_CAPTURE 都是 PHP 4.3.0 之后加入的，而 *offset* 参数则是 PHP 4.3.3 之后加入的，如果你使用的 PHP 版本不够高，可能无法使用这两项功能。

13.3.6 preg_last_error

这个函数返回最近一次调用 preg 系列方法时出现的错误，如果没有出错，则返回 PREG_NO_ERROR，其他错误可以参考“PREG 常量说明”。

13.3.7 preg_replace

mixed preg_replace (mixed $pattern, mixed $replacement, mixed $subject [, int $limit = -1 [, int &$count]])

这个函数用来进行正则表达式替换，将 *subject* 中的 *pattern* 替换为 *replacement*。*pattern* 用于匹配需替换文本的正则表达式，可以是单个字符串，也可以是字符串数组；*replacement* 表示希望替换成的文本，可以在其中引用匹配的相关信息。如果要引用分组，推荐使用${*num*}的记法，这种记法没有二义性，不会引起混淆。

```
preg_replace('/(\d{4})-(\d{2})-(\d{2})/', '${2}/${3}/${1}', "2010-12-20 2011-02-14");
12/20/2010 02/14/2011
```

如果要在 *replacement* 中使用$符号，则必须使用\$转义。

```
preg_replace('/\d+\.\d{0,2}/', '\$${0}', "the price is 12.99");
the price is $12.99
```

subject 表示要进行替换操作的对象，它可以是单个字符串，也可以是字符串数组。如果是字符串，则返回的是字符串；如果是数组，则返回的是数组，同时数组中的每个元素都会进行替换操作。

另外还有两个可选参数，其中 *limit* 在 PHP 4.0.2 以上版本中可用，表示 *pattern* 在 *subject* 中最多能匹配的次数，默认值为-1，表示不限次数；*count* 在 PHP 5.1.0 以上版本中可用，如果传入了这个参数，匹配完成后会将它赋值为匹配实际发生的次数。

13.3.8 preg_replace_callback

```
mixed preg_replace_callback ( mixed $pattern, callback $callback, mixed
$subject [, int $limit = -1 [, int &$count ]] )
```

这个函数的功能类似 preg_replace()，不同的是，它不会简单地把 *pattern* 能匹配的文本替换成其他文本，而是通过 *callback* 函数来处理，这样大大增加了灵活性，*callback* 接收的参数是一个 *match* 数组（而不是字符串），返回字符串。下面的代码借助这个函数，把所有单词统一为首字母大写格式。

```
function capitalize($matches) {
    //正则表达式已经用两个捕获分组分开第一个字母和之后的字母
    return strtoupper($matches[1]).strtolower($matches[2]);
}
preg_replace_callback( '/\b([a-z])([a-z]+)\b/i', "capitalize", "one TWO tHREE");
One Two Three
```

如果觉得这样很麻烦，也可以用create_function()定义匿名函数。[1]

```
preg_replace_callback(    '/\b([a-z])([a-z]+)\b/i',    create_function('$matches',
'return strtoupper($matches[1]).strtolower($matches[2]); '), "one TWO tHREE");
```

在PHP 5.3.0 以上版本中还可以直接定义匿名函数。[2]

```
preg_replace_callback( '/\b([a-z])([a-z]+)\b/i', function($matches) {return
strtoupper($matches[1]).strtolower($matches[2]);} , "one TWO tHREE");
```

13.3.9 preg_filter

```
mixed preg_filter (mixed $pattern, mixed $replacement, mixed $subject [, int
$limit = -1 [, int &$count ]])
```

这个函数类似 preg_replace()，唯一的区别在于，它返回的是确实发生过查找-替换的字符

[1] 可以参考 http://www.php.net/manual/en/function.create-function.php。

[2] 可以参考 http://www.php.net/manual/en/functions.anonymous.php。

串：如果 *subject* 是字符串，且发生了查找-替换，则返回替换之后的字符串，否则返回 NULL；如果 *subject* 是数组，且其中有元素发生了查找-替换，则返回这些发生了查找-替换的元素构成的数组，否则返回空数组。

```
$textArray = array("2010-12-20", "other text", "2011-02-14");
preg_filter('/(\d{4})-(\d{2})-(\d{2}/)', "$2/$3/$1", $ textArray);

Array{ [0]=>"12/20/2010", [1]=>"02/14/2011" }
```

PHP 5.2.0 之后提供了另一个函数 filter_var()，它通过预定义的过滤器来判断字符串的值，进行常用的过滤，而不需要显式用到正则表达式，如果判断成功则返回字符串，否则返回 false。常见的过滤器包括：

FILTER_VALIDATE_EMAIL：验证是否为 E-mail
FILTER_VALIDATE_FLOAT：验证是否为浮点型
FILTER_VALIDATE_INT：验证是否为整型
FILTER_VALIDATE_IP：验证是否是合法 IP
FILTER_VALIDATE_URL：验证是否是合法 URL

以下是示例代码：

```
var_dump(filter_var("one@example....com", FILTER_VALIDATE_EMAIL));
false

var_dump(filter_var("one@example.com", FILTER_VALIDATE_EMAIL));
one@example.com
```

13.3.10 preg_split

array preg_split(string $pattern, string $subject [, int $limit = -1 [,int $flags = 0]])

这个函数用来切分字符串，返回切分所得的字符串数组。*pattern* 匹配的文本作为切分的分隔，而 *subject* 则是切分的字符串。

```
preg_split('/\s+/', "one\ttwo\n three");
Array{ [0]=>"one", [1]=>"two", [2]=>"three" }
```

limit 是可选参数，它设定 *pattern* 匹配的次数。如果设定了 *limit*，那么返回数组包含 *limit* 个元素，也就是说，最多进行 *limit*-1 次切分。如果 *limit* 设为-1、0、null，都表示“不设定上限”，也就是“切分尽可能多”。

```
preg_split('/\s+/', "one\ttwo\n three", -1);
```

```
Array{ [0]=>"one", [1]=>"two", [2]=>"three" }

preg_split('/\s+/', "one\ttwo\n three", 2);
Array{ [0]=>"one", [1]=>"two  three" }
```

flags 是可选参数，用于控制返回的数据，还可以选择常量 PREG_SPLIT_NO_EMPTY，它要求返回数组忽略空字符串。

```
preg_split('/,/', "one,two,,three", -1);
Array{ [0]=>"one", [1]=>"two", [2]=>"", [3]=>"three"}

preg_split('/,/', "one,two,,three", -1, PREG_SPLIT_NO_EMPTY);
Array{ [0]=>"one", [1]=>"two", [2]=>"three"}
```

有可能引起混淆的是 *limit* 和 PREG_SPLIT_NO_EMPTY 同时作用的情况，*limit* 设定的并不是“返回数组的长度”，而是“*pattern* 匹配的次数”，所以并不会“先全部切分，再过滤数组”。

```
preg_split('/,/', "one,two,,three", 2, PREG_SPLIT_NO_EMPTY);
Array{ [0]=>"one", [1]=>"two", [2]=>",three"}
```

13.3.11　preg_replace_callback_array

```
mixed preg_replace_callback_array ( array $patterns_and_callbacks , mixed
$subject [, int $limit = -1 [, int &$count ]] )
```

这是PHP 7新增的函数，功能与preg_replace_callback()类似。在preg_replace_callback()中，可以定义一个函数来对匹配结果进行复杂的替换操作，但有些时候单个函数仍然不够，我们希望在替换操作中进行更复杂的操作，preg_replace_callback_array()就可以做到这点，它传入一组函数，可以对匹配结果进行分门别类的处理。下面的代码借助这个函数，把所有 T 字母开头的单词变为首字母大写，其他字母开头的单词变为全大写。

```
preg_replace_callback(
    [
        '/\b([t])([a-z]+)\b/i' => function ($match) {
            return strtoupper($matches[1]).strtolower($matches[2]);
        },
        '/\b([a-su-z])([a-z]+)\b/i' => function ($match) {
            return strtoupper($matches[0]);
        }
    ],
    "one TWO tHREE");
ONE Two Three
```

13.4 常见的正则操作举例

13.4.1 验证

```
//记得要在正则表达式首尾添加\A 和\z
$regex = '/\A\d{4}-\d{2}-\d{2}\z/';
preg_match($regex, "2010-12-20");    // => True
```

13.4.2 提取

逐个提取

```
$matches = array();
$offset = 0;
$regex = '/\d{4}-\d{2}-\d{2}/';
$string = "2010-12-20 2011-02-14";
while (preg_match($regex, $string, $matches, PREG_OFFSET_CAPTURE, $offset)!=0 ) {
        echo($matches[0][0]);
        echo "<br />";
        $offset = $matches[0][1] + strlen($matches[0][0]);
}
2010-12-20
2011-02-14

$matches = array();
$offset = 0;
$regex = '/(\d{4})-(\d{2})-(\d{2})/';
$string = "2010-12-20 2011-02-14";
while (preg_match($regex, $string, $matches, PREG_OFFSET_CAPTURE, $offset)!=0 ) {
    echo "date: ".$matches[0][0].", year: ".$matches[1][0].", month: ".$matches[2][0].", day:
".$matches[3][0];
        echo "<br />";
        $offset = $matches[0][1] + strlen($matches[0][0]);
}
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

一次性提取

```
//为符合习惯，均设定 PREG_SET_ORDER 以每次匹配为单位
$matches = array();
$regex = '/\d{4}-\d{2}-\d{2}/';
$string = "2010-12-20 2011-02-14";
```

```
preg_match_all($regex, $string, $matches, PREG_SET_ORDER) ;
foreach ($matches as $match) {
    echo($match[0]);
}
2011-03-17
2010-12-24

$matches = array();
$offset = 0;
$regex = '/(\d{4})-(\d{2})-(\d{2})/';
$string = "2010-12-20 2011-02-14";
preg_match_all($regex, $string, $matches, PREG_SET_ORDER) ;
foreach ($matches as $match) {
    echo("date: ".$match[0].", year: ".$match[1].", month: ".$match[2].", day:
".$match[3]);
    echo("<br />");
}
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

13.4.3　替换

简单替换

```
$regex = '/(\d{4})-(\d{2})-(\d{2})/';
$string = "2010-12-20 2011-02-14";
$replacement = '${2}/${3}/${1}';
preg_replace($regex, $replacement, $string);
12/20/2010 02/14/2011
```

在 replacement 中使用$

```
$regex = '/\d+\.\d{0,2}/';
$replacement = '\$${0}';
$string = "the price is 12.99";
preg_replace($regex, $replacement, $string);
the price is $12.99
```

13.4.4　切分

```
$regex = '/-/';
$string = "2010-12-20";
preg_split($regex, $string);
Array { [0]=>"2010", [1]=>"12", [2]=>"20" }
```

第 14 章　Python

Python对正则表达式的支持较为完备，它的API很丰富，也支持Unicode字符串（Python 2 的字符串并没有默认使用Unicode编码，所以使用上稍微麻烦一点），但不支持Unicode属性（☞127）。因为Python 2 和Python 3 在正则表达式的规定上大不相同，本章的讲解主要基于Python 2（具体版本是Python 2.7.14），示例代码通常也可以用于Python 3（作者基于Python 3.6.4 进行了全部测试），遇到区别会专门讲解。[1]

14.1　预备知识

Python 内置了正则表达式的 package，在使用正则表达式之前，必须首先导入对应的 package：

```
import re
```

为讲解方便，先介绍 Python 中的 `re.search(pattern, string)`函数。如果 *pattern* 可以匹配 *string* 的某个子串，则返回一个 `MatchObject` 对象，否则返回 `None`。通过判断是否返回 `None`，就可以知道 *pattern* 能否匹配 *string* 中的子串。如果在正则表达式两端加上`\A`和`\Z`（注意是`\Z`而不是`\z`，Python 中没有`\z`，`\Z`等价于其他语言中的`\z`），则可以判断整个字符串是否能由正则表达式匹配，进行正则表达式验证。

```
re.search("\\d{6}", "123456") != None         # => True
re.search("\\d{6}", "12345678") != None       # => True
re.search("\\d{6}", "a123456") != None        # => True
re.search("\\A\\d{6}\\Z", "123456") != None   # => True
re.search("\\A\\d{6}\\Z", "12345678") != None # => False
re.search("\\A\\d{6}\\Z", "a123456") != None  # => False
```

Python 的字符串比较复杂，值得花点篇幅进行介绍：Python 中常用的字符串有单引号字符串和双引号字符串，两者并没有区别，`"\\d{6}"`与`'\\d{6}'`是完全等价的。如果使用这两种字符串来表示正则表达式，都必须经过字符串转义与正则转义（详见第 6 章），也就是说，正则表达

[1] Python 2 和 Python 3 的一大差别在于，print 在 Python 3 中是函数，所以 print text 之类的写法在 Python 3 中不再可行，而应当改为 print(text)。请读者阅读时注意。

式\d应当表现为字符串文字\\d。

但是 Python 也提供了另一种“原生字符串（Raw String）”，其方式是在字符串之前添加字符r，这样就会忽略字符串转义，只需要提供正则文字即可。这样，正则表达式\d就可以写作r"\d{6}"或r'\d'。这种形式更适合正则表达式，所以本章中的正则表达式都使用这种形式。

```
re.search(r"\A\d{6}\Z", "123456") != None     # => True
re.search(r"\A\d{6}\Z", "12345678") != None   # => False
re.search(r"\A\d{6}\Z", "a123456") != None    # => False
```

你有可能在某些代码中看到正则表达式\d直接使用字符串文字\d，而没有使用原生字符串，但也并没有出错。这是因为 Python 在处理字符串转义时，如果字符串文字中的某个转义序列无法识别，则会被原封不动地“保留下来”。\d 并不是可以识别的字符串转义序列，所以会被保留下来，作为正则文字。因此\d也可以直接写成字符串\d，同样\A和\Z也是如此（☞95）。

```
re.search("\A\d{6}\Z", "123456") != None      # => True
re.search("\A\d{6}\Z", "12345678") != None    # => False
re.search("\A\d{6}\Z", "a123456") != None     # => False
```

但我并不推荐使用这种做法，因为表达式\b不能写成"\b"——因为在字符串文字中\b 是可以识别的字符串转义序列，表示退格符（backspace），所以"\bcat\b"生成的正则表达式不能匹配“单词 cat”，而是匹配“两个退格符之间的 cat”。

```
re.search("\bcat\b", "cat") != None       # => False
re.search("\\bcat\\b", "cat") != None     # => True
re.search(r"\bcat\b", "cat") != None      # => True
```

另外，Python 2 的字符串并非默认采用Unicode编码，所以如果要用到Unicode功能（并非Unicode属性）[1]，则必须在字符串之前添加字符u，比如u"[你我他]"。所以，如果字符串中出现了多字节字符，最好使用u属性，写成u"你"；如果正则表达式中出现了多字节字符，最好是同时使用u和r。不过，注意，只能写作ur"[你我他]", 不能写作ru"[你我他]"。而在Python 3 中，因为字符串默认采用Unicode编码，所以只需用r标记，比如r"[你我他]"。

14.2　正则功能详解

14.2.1　列表

表 14-1 列出了 Python 中的正则功能。

[1] Python 2 中并没有“Unicode 字符串”，而只有“Unicode 对象”，所谓“Unicode 字符串”也只是被当作 str 的“Unicode 对象”而已。而 str 对象只是一个字符数组，里面存储什么内容，使用什么编码，都需要用户自己负责。所以，如果在 Windows 的 Python IDLE（而不是命令行提示符）下使用 Python，即便指定了 u，也生成不了真正的 Unicode 字符串，测试本章中的例子时可能会出错。

表 14–1 Python 中的正则功能

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	支持 Unicode
POSIX 字符组 ☞15	`[:digit:]`	只匹配 ASCII 字符
Unicode 属性 ☞127	`\p{…}`	不支持
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	2.x 中默认采用 ASCII 匹配规则，但可设定 Unicode 模式变为 Unicode 匹配规则；3.x 中默认使用 Unicode 模式，但可设定 ASCII 模式变为 ASCII 匹配规则
行起始位置 ☞62	`^` `\A`	支持
行结束位置 ☞62	`$` `\z` `\Z`	不支持`\z`
单词边界 ☞123	`\b` `\B`	2.x 中默认采用 ASCII 匹配规则，但可设定 Unicode 模式变为 Unicode 匹配规则；3.x 中默认使用 Unicode 模式，但可设定 ASCII 模式变为 ASCII 匹配规则
顺序环视 ☞69	`(?=…)` `(?!…)`	支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	不支持变长表达式
匹配模式 ☞83	`i m s x a u`	支持
模式作用范围 ☞91	`(?`*modifier*`)` `(?-`*modifier*`)`	只支持`(?`*modifier*`)`
纯文本模式 ☞101	`\Q…\E`	不支持
捕获分组及引用☞44	`(…)` `\`*num* $*num*	支持，同时支持命名分组
命名分组 ☞53	`(?P<`*name*`>…)` `(?P=`*name*`)`	支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

14.2.2 字符组

14.2.2.1 Python 2

除非显式指定，否则 Python 2 的字符串默认都采用 ASCII 编码，正则表达式也无法正确识别 GBK 编码环境下的多字节字符，因此可能遇到错误匹配的问题（☞116）。

```
re.search(r"\A.\Z", "发") != None        # => False
```

```
re.search(r"[正则]", "遭") != None        # => True
```

解决的办法是使用 Unicode 编码环节。请注意，不管是用作正则表达式的字符串，还是被匹配的字符串，只要出现了中文字符，都必须指定 u。

```
re.search(r"\A.\Z", u"发") != None        # => True
re.search(ur"[正则]", u"遭") != None      # => False
```

14.2.2.2　Python 3

Python 3 中的字符串默认采用 Unicode 编码，所以不存在上面的问题。

14.2.2.3　关于编码

如果不清楚自己使用的 Python 采用什么编码，可以运行下面的代码得到。

```
import sys
sys.getdefaultencoding()
```

如果显示 ASCII，则会遇到 GBK 编码的错误；如果显示为 UTF-8，则不会。

有时，Python 可能会提示程序文件中出现的 UTF-8 字符无法识别，这时应该在.py 文件的开头加上这样一行，告诉解释器，本源文件使用的是 UTF-8 编码：

```
# encoding=utf-8
```

在 Mac OS 和 Linux/UNIX 下，Python 会自动从环境变量中获得编码（通过设定 LANG=en_US.UTF-8 或 LANG=zh_CN.UTF-8）。Windows 默认使用的并不是 Unicode 编码。所以，如果要在 Windows 下使用 Python，又要使用 UTF-8 编码，除了直接使用 Python 3，Python 2 必须在.py 文件的顶部加上这一行。

在 Python 2 下，还有一个推荐的做法，一次性将所有字符串都指定为 Unicode 字符串：

```
from __future__ import unicode_literals
```

这样确实省事，但仍然可能遇到一些问题。因为 Python 2 的不少函数只接受 str 类型的字符串，不接受 Unicode 字符串。解决办法如下面的代码所示：

```
from __future__ import unicode_literals
"不能接受的 Unicode 字符串".encode("utf-8")
```

14.2.3　Unicode 属性

Python 2 和 Python 3 都不支持 Unicode 属性，也就是说，所有规定的 Unicode 属性都无法在 Python 中使用。如果一定要用 Unicode 属性，可以使用第三方库，比如 Ponyguruma。具体请读者自行搜索，此处不做详细介绍。

14.2.4 字符组简记法

14.2.4.1 Python 2

在 Python 2 中，在默认情况下，\d、\D、\w、\W、\s、\S 都采用字符组简记法的 ASCII 匹配规则（☞121）。也就是说，\d 等价于[0-9]，\w 等价于[0-9a-zA-Z_]，\s 无法匹配其他空白字符（比如中文的全角空格）。

```
#半角字符
re.search(r"\d", "1") != None    # => True
re.search(r"\w", "a") != None    # => True
re.search(r"\s", " ") != None    # => True

#全角字符及中文字符
re.search(r"\d", u"１") != None  # => False
re.search(r"\w", u"我") != None  # => False
re.search(r"\s", u"　") != None  # => False
```

但是如果启用 Unicode 匹配模式（最简单的做法是在表达式的最前面加上(?u)），则会启用字符组简记法的 Unicode 匹配规则。

```
#全角字符及中文字符
re.search(r"(?u)\d", "１") != None   # => True
re.search(r"(?u)\w", "我") != None   # => True
re.search(r"(?u)\s", "　") != None   # => True
```

目前已经出现了两个概念：“Unicode 字符串”和“Unicode 模式”，它们比较容易混淆，表 14-2 有助于澄清这两个概念。

表 14–2 Python 中的 Unicode 字符串和 Unicode 模式

名称	Unicode 字符串	Unicode 模式
使用对象	字符串	正则表达式
作用	正确识别多字节字符，而不是将其割裂为多个字节	扩展字符组简记法中“数字”“空白”“单词字符”的意义
形式	在字符串前面加上 u，比如 u"我"	在表达式开头加上(?u)，比如 u"(?u)\w"
应用场景	字符串中出现了汉字	需要\d、\D、\w、\W等简记法识别中文字符
可否连用	可以，但必须记为 ur，不能记为 ru	

14.2.4.2 Python 3

在 Python 3 中，各字符组简记法默认都采用字符组简记法的 Unicode 匹配规则（☞120）。也就是说，在默认情况下，\d 能匹配 Unicode 的数字字符，\w 能匹配 Unicode 的单词字符，\s

匹配 Unicode 的空白字符。

```
#半角字符
re.search(r"\d", "1") != None    # => True
re.search(r"\w", "a") != None    # => True
re.search(r"\s", " ") != None    # => True

#全角字符及中文字符
re.search(r"\d", "１") != None   # => True
re.search(r"\w", "我") != None   # => True
re.search(r"\s", "　") != None   # => True
```

如果要“还原”为 ASCII 匹配规则，则必须显式设定 ASCII 模式（最简单的做法是在表达式最前面加上`(?a)`）。

```
#全角字符及中文字符
re.search(r"(?a)\d", "１") != None   # => False
re.search(r"(?a)\w", "我") != None   # => False
re.search(r"(?a)\s", "　") != None   # => False
```

在 Python 2 和 Python 3 中，默认字符组简记法的匹配规则完全相反，使用时必须尤其注意。比如在 Python 2 中完全可以用`^\d+$`验证 ASCII 字符数字的字符串，但在 Python 3 中则不能这样做。

14.2.5　单词边界

14.2.5.1　Python 2

Python 2 的正则表达式中单词边界的规定与对`\w`的规定是一样的。也就是说，`\b`匹配成功只有一种情况：一边必须保证`\w`（等价于`[0-9a-zA-Z_]`）匹配成功，另一边必须保证`\w`匹配不成功。

```
#空字符串-英文字符
re.search(r"a\b", "a") != None       # => True
#英文字符-半角标点
re.search(r"a\b", "a,") != None      # => True
#英文字符-半角空格
re.search(r"a\b", "a ") != None      # => True
#英文字符-中文字符
re.search(r"a\b", "a我") != None     # => True
#英文字符-全角空格
re.search(r"a\b", "a　") != None     # => True
#中文字符-半角标点
re.search(r"我\b", "我,") != None    # => False
#中文字符-半角空格
```

```
re.search(r"我\b", "我 ") != None     # => False
#中文字符-中文字符
re.search(r"我\b", "我和") != None    # => False
#中文字符-全角空格
re.search(r"我\b", "我　") != None    # => False
```

如果显式指定了 Unicode 匹配模式，则\w能匹配的字符包括中文字符，所以\b的匹配也相应变化。

```
#英文字符-中文字符
re.search(r"(?u)a\b", "a我") != None     # => False
#中文字符-半角标点
re.search(r"(?u)我\b", "我,") != None     # => True
#中文字符-半角空格
re.search(r"(?u)我\b", "我 ") != None     # => True
#中文字符-全角空格
re.search(r"(?u)我\b", "我　") != None    # => True
```

14.2.5.2 Python 3

因为 Python 3 默认采用 Unicode 匹配规则，默认情况下，\w认定的单词字符就包括其他语言中的字符(包括中文字符)，等价于 Python 2 显式指定 Unicode 模式的情况。如果希望使用 ASCII 匹配规则，则应当显式指定 ASCII 模式，它等价于 Python 2 中的默认情况。

14.2.6 行起始/结束位置

^匹配的是“行开始的位置”，在默认情况下，它只能匹配“整个字符串的开始位置”，如果指定使用多行模式（Multiline Mode），^可以匹配字符串内部的文本行的开始位置；而\A无论在什么情况下都匹配“整个字符串的开始位置”。

```
re.search(r"^1", "1\n2\n") != None       # => True
re.search(r"^2", "1\n2\n") != None       # => False
re.search(r"(?m)^2', "1\n2\n") != None   # => True

re.search(r"\A1", "1\n2\n") != None      # => True
re.search(r"\A2", "1\n2\n") != None      # => False
re.search(r"(?m)\A2", "1\n2\n") != None  # => False
```

在默认情况下，Python 中的$和\Z匹配的是整个字符串的结束位置(如果结束位置有换行符，则$匹配这个换行符之前的位置)，使用多行模式会影响$的匹配，这时候它可以匹配文本内部行的结束位置。所以如果要进行严格准确的验证，应当使用\Z，它等价于其他语言中的\z。

```
re.search(r"1$/", "1\n") != None         # => True
re.search(r"1$/", "1\n2") != None        # => False
re.search(r"(?m)1$ ", "1\n2") != None    # => True
```

```
re.search(r"1\Z", "1") != None          # => True
re.search(r"1\Z", "1\n") != None        # => False
re.search(r"(?m)1\Z", "1\n2") != None   # => False
```

14.2.7　环视

Python 中的肯定顺序环视`(?=…)`和否定顺序环视`(?!…)`内可以使用任意形式的子表达式。

```
re.search(r"c(?=ab+)", "cab")           # => True
re.search(r"c(?=(ab+|cd))", "ccd")      # => True
re.search(r"c(?=(ab+|cd))", "cabbbb")   # => True
```

Python 中的逆序环视则多了很多限制：在逆序环视`(?<=…)`和`(?!<…)`中，表达式能匹配的文本长度必须是固定的，否则就会报错。也就是说，逆序环视中的表达式不能使用`+`、`*`、`?`之类的量词；如果出现多选结构，各个多选分支能匹配的文本长度也必须是相同的。

```
re.search(r"ab(?<=ab)", "ab")           # => True
re.search(r"cd(?<=ab?)", "cd")          #报错，ab?能匹配的文本长度不确定
re.search(r"cd+(?<=(abcd|cd))", "cd")   #报错，abcd 和 cd 匹配的文本长度不同
```

14.2.8　匹配模式

Python 的正则表达式支持所有常见的模式，另外还提供了 Unicode 模式和 ASCII 模式，详见表 14-3。

表 14–3　Python 中的匹配模式

常量	修饰符	解释
re.I re.IGNORECASE	i	不区分大小写模式
re.X re.VERBOSE	x	注释模式
re.M re.MULTILINE	m	多行模式
re.S re.DOTALL	s	单行模式
re.U re.UNICODE	u	Unicode 模式
re.A re.ASCII	a	ASCII模式 [1]

[1] re.A 和 re.ASCII 模式只有在 Python 3 中才有。

匹配模式可以在表达式中以(?…)指定，也可以将常量作为参数指定。

```
re.search("a", "A") != None                  # => False
re.search("(?i)a", "A") != None              # => True
re.search("a", "A", re. IGNORECASE) != None  # =>True
```

需要注意的是，Python 中的模式永远是对**整个正则表达式**生效的，无论(?…)出现在哪里。

```
re.search("a(?i)b", "AB") != None # => True
```

Python 不支持用(?-…)停用模式。

14.2.9 捕获分组的引用

在 Python 中，如果要在正则表达式内部引用捕获分组，则应当使用*num* 记法，其中 *num* 为对应捕获分组的编号。

```
re.search(r"([a-z])\1", "ab") != None    # => False
re.search(r"([a-z])\1", "aa") != None    # => True
```

如果要在替换时引用捕获分组，则仍然使用*num* 记法。

```
re.sub(r"(\d{4})-(\d{2})-(\d{2})", r"\2/\3/\1", "2010-12-20")
12/20/2010
```

更好的办法是使用\g<*num*>，这样更加清晰，避免了二义性。

```
re.sub(r"(\d{4})-(\d{2})-(\d{2})", r"\g<2>/\g<3>/\g<1>", "2010-12-20")
12/20/2010
```

如果使用了命名分组，在表达式中应当使用(?P=*name*)来引用，在替换时应当使用\g<*name*>来引用。

```
re.search(r"(?P<char>[a-z])(?P=char)", "ab") != None   # => False
re.search(r"(?P<char>[a-z])(?P=char)", "aa") != None   # => True
```

```
regex = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})";
replacement = r"\g<month>/\g<day>/\g<year>";
print re.sub(regex, replacement, "2010-12-20")


12/20/2010
```

14.2.10 条件匹配

Python 的正则表达式支持条件匹配，这是一个很有趣的功能，值得详细介绍一下。

在正则表达式的匹配过程中，我们有时候会希望进行更细致的操控：假如前面这样匹配了，后面要这样匹配；假如前面那样匹配了，后面要那样匹配。但是，经典的正则表达式语法并没有

提供这样的功能，往往只能靠多个正则表达式加上程序代码来实现。Python 的正则引擎直接支持条件匹配，使用起来就方便很多了。

条件匹配的语法是这样的：`(?(id/name)yes-pattern|no-pattern)`。其中 *id/name* 是对应捕获分组的名称或者编号，如果该捕获分组成功匹配文本，则后续匹配交由 *yes-pattern* 来完成，否则交由 *no-pattern* 来完成。*no-pattern* 不是必须出现的，可以省略。假设我们需要验证价格，规则如下：如果前面没有美元符号$，则价格只能包含整数部分，否则还应当包含小数点和两位小数。条件匹配正适合解决这个问题。

```
regex = ur"^(\$)?[0-9]+(?(1)\.[0-9]{2}|)$"
re.search(regex, u"34") != None              # => True
re.search(regex, u"12.00") != None           # => False
re.search(regex, u"$34") != None             # => False
re.search(regex, u"$12.00") != None          # => True
```

这个例子看起来有些简单，确实，通常我们都会在正则表达式中使用多选分支，也就是`^(\$[0-9]+\.[0-9]{2}|[0-9]+)$`，可起到同样的效果。但是如果两个多选分支中相同的部分是一个复杂的表达式（而不是本例中的`[0-9]+`），那么条件匹配会让表达式简单很多。

另外需要特别注意的是，条件匹配成立的条件是“对应捕获分组成功匹配文本”，所以对应捕获分组通常需要用量词来修饰。即便“对应捕获分组捕获到空文本”，也不等于“对应捕获分组匹配文本失败”，所以不会尝试 *no-pattern* 的匹配。

14.3　正则 API 简介

14.3.1　RegexObject

Python 中的正则表达式对象是 `RegexObject`，它可以由正则表达式编译而来，常用的方法是 `re.compile(`*pattern*`)`。

```
regexObject = re.compile("ab+")
```

如果要指定匹配模式，则可以在表达式中使用模式修饰符，也可以使用预定义常量。

```
regexObject = re.compile("(?i)ab+")
regexObject = re.compile("ab+", re.I)
```

如果要同时指定多个匹配模式，则可以在表达式中连写模式修饰符，也可以用|运算符连接预定义常量。

```
regexObject = re.compile((?is)"ab+")
regexObject = re.compile("ab+", re.I|re.S)
```

在 Python 的文档中，大多数函数是类似 re.search(*pattern, string*)的形式，其中的 *pattern* 通常是正则表达式的字符串，可以将其视为“静态方法”。其实也可以将 RegexObject 对象作为 *pattern* 参数传入，或者直接对 RegexObject 对象调用相同名字的成员方法，结果相同，但是这两种做法能显式缓存 RegexObject 对象。下面的代码分别使用了三种办法，结果相同。

```
re.search(r"\d{6}", "123456") != None         # => True

sixDigitRegex = re.compile(r"\d{6}")
sixDigitRegex.search("123456") != None        # => True

sixDigitRegex = re.compile(r"\d{6}")
re.search(sixDigitRegex, "123456") != None    # => True
```

re.compile()还有一个不为许多人所知的功能，就是它可以用来观察某个正则表达式的详细信息，方法是指定第二个参数为 re.DEBUG。如果你遇到复杂的表达式，或者不确定某个表达式的意义，可以通过它来观察。下面给出两个简单的例子，其中的缩进表示了表达式各结构的层级关系，而字符本身则显示为其码值的十进制表示（比如字符 a 的码值是十进制的 97）。

```
re.compile("ab|cd", re.DEBUG)
branch
  literal 97
  literal 98
or
  literal 99
  literal 100

re.compile("(ab|[cde])+", re.DEBUG)
max_repeat 1 65535
  subpattern 1
    branch
      literal 97
      literal 98
    or
      in
        literal 99
        literal 100
        literal 101
```

14.3.2 re.compile(regex[, flags])

这个方法用于显式“编译”正则表达式对象。如果需要多次使用同一个正则表达式，那么，每次使用编译好的正则表达式对象，比每次临时编译表达式要快得多。re.compile()的实际使用在上一节已经看到过，这里不再重复。

需要指出的是，Python 会自动缓存最近一次在 re.match()、re.search()和 re.compile()中指定的正则表达式。如果用到的正则表达式并不多，可以使用字符串形式的正则表达式，而不必显式调用 re.compile()获得 RegexObject 对象。也正因为如此，本章的大部分例子都没有显式调用 re.compile()，而是直接以字符串方式给出正则表达式。在实际编程中，可以根据具体情况决定是否显式调用 re.compile()。

14.3.3　re.search(pattern, string[, flags])

这个方法用来测试正则表达式 *pattern* 能否在 *string* 中找到匹配，*flags* 是可选参数，表示匹配模式（下面各个方法中的 *flags* 参数都具有同样的功能，不再赘述）。如果能找到匹配，则返回 MatchObject（详见下文），否则返回 None。

```
re.search(r"\d", "ab")      # 匹配不成功
re.search(r"\d", "ab1")     # 得到一个 MatchObject
<_sre.SRE_Match object at 0x0000000001D55E00>
```

这个函数可以用来进行数据验证，在正则表达式两端加上\A 和\Z（Python 不支持\z，\Z 等价于其他语言中的\z），并判断返回值是否为 None。

```
re.search(r"\A\d{6,10}\Z", "1234567ab") != None    # => False
re.search(r"\A\d{6,10}\Z", "1234567") != None      # => True
```

14.3.4　MatchObject

调用 re.search()或者 re.match()，如果匹配成功，就返回一个 MatchObject 对象。实际上，与正则表达式匹配有关的许多方法都会返回 MatchObject 对象。

MatchObject 提供了若干方法和属性，通过它们可以获得匹配的信息，表 14-4 列出了常用的方法和属性。

表 14–4　MatchObject 常用的方法和属性

方法	描述	注释
start(*n*)	返回编号为 *n* 的捕获分组匹配的开始位置，如果对应分组不存在或未匹配，则返回-1	如果没有设定 *n*，则返回整个表达式匹配的开始位置，此时 *n* 等价于 0
pos	返回整个表达式匹配的开始位置	
end(*n*)	返回编号为 *n* 的捕获分组匹配的结束位置，如果对应分组不存在或未匹配，则返回-1	如果没有设定 *n*，则返回整个表达式匹配的结束位置，此时 *n* 等价于 0
endpos	返回整个表达式匹配的结束位置	等价于 end(0)
group(*n*)	返回编号为 *n* 的捕获分组匹配的文本	如果没有设定 *n*，则返回整个表达式匹配的文本，此时 *n* 等价于 0

（续表）

方法	描述	注释
groups()	返回各捕获分组匹配的文本构成的元组	如果表达式中不存在捕获分组，则返回空元组
span(*n*)	返回编号为 *n* 的捕获分组匹配的开始-结束位置	如果没有设定 *n*，则返回整个表达式匹配的开始-结束位置，此时 *n* 等价于 0
re	返回匹配时所使用的正则表达式对象	
lastindex	返回匹配成功的编号最大的捕获分组的编号	如果没有，则返回 None
string	返回用来尝试用正则表达式来匹配的字符串	
expand(*str*)	返回一个字符串，可以在其中引用捕获分组匹配的文本	

下面的代码集中示范了其中主要的方法。

```
obj = re.search(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-20")
print "%s starts at %d and ends at %d" % (obj.group(), obj.pos, obj.endpos)
for i in range(1, obj.lastindex+1) :
    print "%s starts at %d and ends at %d" % (obj.group(i), obj.start(i), obj.end(i))
print obj.expand(r"year:\1 month:\2 day:\3")
2010-12-20 starts at 0 and ends at 10
2010 starts at 0 and ends at 4
12 starts at 5 and ends at 7
20 starts at 8 and ends at 10
year:2010 month:12 day:20
```

14.3.5 re.match(pattern, string[, flags])

re.match()和 re.search()非常相似，参数相同，返回值也相同，非常容易混淆，唯一的区别在于：如果 re.match()匹配成功，那么 ***pattern*** 匹配的字符串必须开始于 ***string*** 的最左端。也就是说，re.match()只会在字符串的开始位置尝试匹配，re.search()则没有这个限制。

```
re.search(r"\d", "ab1") != None      # => True
re.match(r"\d", "ab1") != None       # => False
re.search(r"\d", "1ab") != None      # => True
re.match(r"\d", "1ab") != None       # => True
```

re.match()的名字容易让人认为这是专门用来进行数据验证的方法。其实单纯调用它并不能准确验证，还必须在正则表达式末尾加上\Z，不过更好的做法是在正则表达式首尾加上\A 和\Z，这样更符合惯例。

```
re.search(r"\d{6}", "123456") != None         # => True
#错误的验证
```

```
re.search(r"\d{6}", "123456ab") != None       # => True
#尾部加上\Z
re.search(r"\d{6}\Z", "123456ab") != None     # => False
#推荐的做法
re.search(r"\A\d{6}\Z", "123456ab") != None   # => True
```

14.3.6　re.findall(pattern, string[, flags])

这个函数会一次性找出正则表达式 *pattern* 在 *string* 中的所有匹配，返回一个列表，最简单的用法就是：

```
re.findall(r"\d{4}-\d{2}-\d{2}", "2010-12-20 2011-02-14")
['2010-12-20', '2011-02-14']
```

如果 *pattern* 中存在捕获分组，则返回列表的元素并非整个表达式所匹配的文本，而是各个捕获分组所匹配文本构成的元组。

```
re.findall(r"(\d{4})-(\d{2})-(\d{2})", "2010-12-20 2011-02-14")
[('2010', '12', '20'), ('2011', '02', '14')]
```

请注意，如果 *pattern* 中存在捕获分组，整个表达式匹配的文本不会包含在返回文本中，而且返回元组中的元素偏移值与分组编号并不对应（编号为 1 的分组匹配的文本是元组中的元素 0，编号为 2 的分组匹配的文本是元组中的元素 1，依此类推）。如果需要得到整个表达式匹配的文本，则可以显式给整个表达式增加括号。

```
re.findall(r"((\d{4})-(\d{2})-(\d{2}))", "2010-12-20 2011-02-14")
[('2010-12-20', '2010', '12', '20'), ('2011-02-14', '2011', '02', '14')]
```

因为它返回的是列表而不是字典，所以即便使用了命名分组，`re.findall()`也只会将它们当作数字编号分组来处理，而不会保存任何与命名有关的信息。

```
regex = r"((?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2}))"
re.findall(regex, "2010-12-20 2011-02-14")
[('2010-12-20', '2010', '12', '20'), ('2011-02-14', '2011', '02', '14')]]
```

14.3.7　re.finditer(pattern, string[, flags])

如果文本内容很多，使用 `re.findall()`一次性找出所有匹配的成本可能很高，使用 `re.finditer()`可以返回一个迭代器（iterator），它可以按从左到右的顺序逐个遍历每次匹配的 `MatchObject` 对象，依次进行处理或检验。

```
#看起来是先一次性找到所有匹配结果，再遍历，其实每次的匹配结果是临时得到的
regex = r"(\d{4})-(\d{2})-(\d{2})"
for iter in re.finditer(regex, "2010-12-20 2011-02-14"):
    print "%s, %s, %s, %s" % (iter.group(), iter.group(1), iter.group(2), iter.
group(3))
```

```
2010-12-20, 2010, 12, 20
2011-02-14, 2011, 02, 14
```

```
#使用命名分组
regex = r"((?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2}))"
for iter in re.finditer(regex, "2010-12-20 2011-02-14"):
    print "%s, %s, %s, %s" % (iter.group(), iter.group(1), iter.group(2),
iter.group(3))
```

```
2010-12-20, 2010, 12, 20
2011-02-14, 2011, 02, 14
```

14.3.8 re.split(pattern, string[, maxsplit=0, flags=0])

这个函数用正则表达式 *pattern* 能够匹配的文本切分字符串 *string*。

```
re.split(r"-", "2010-12-20")
['2010', '12', '20']
```

如果正则表达式在字符串的开头或结尾位置能找到匹配，那么结果的开头或结尾会出现空字符串。

```
re.split(r"-", "-2010-12-20-")
['', '2010', '12', '20', '']
```

也可以明确设定 *maxsplit*。请注意，在其他语言中，这个参数一般表示返回数组的长度，但是在 Python 中，它表示切分的次数。也就是说，返回数组一般会包含 *maxsplit*+1 个元素。如果将 *maxsplit* 设定为负数，则表示“不做任何切分”；若设定为 0，则等于没有指定；若设定的值为正数，且小于实际可切分次数，则按设定的值进行切分，若大于实际可切分次数，则以实际可切分次数为准。

```
re.split(r"-", "2010-12-20", -1)
['2010-12-20']
re.split(r"-", "2010-12-20", 0)
['2010', '12', '20']
re.split(r"-", "2010-12-20", 1)
['2010', '12-20']
```

14.3.9 re.sub(pattern, repl, string[, count, flags])

这个函数用来进行正则表达式替换，它将字符串 *string* 中正则表达式 *pattern* 能够匹配的文本替换为 *repl* 指定的文本。

```
regex = r"\d{4}-\d{2}-\d{2}"
replacement = "Date"
re.sub(regex, replacement, "2010-12-20 2011-02-14")
Date Date
```

如果要在 *replacement* 字符串中引用之前某个捕获分组匹配的文本，则应当使用*num*，不过因为 *replacement* 是字符串，如果使用字符串文字，则应当写成*num*。为了清晰直接，推荐使用原生字符串，也就是在字符串之前添加 r。

```
regex = r"(\d{4})-(\d{2})-(\d{2})"
replacement = r"\2/\3/\1"
print re.sub(regex, replacement, "2010-12-20")
12/20/2010
```

Python 也提供了更清晰的方式在 *repl* 中引用分组，写作\g<*num*>。

```
regex = r"(\d{4})-(\d{2})-(\d{2})"
replacement = r"\g<2>/\g<3>/\g<1>"
print re.sub(regex, replacement, "2010-12-20")
12/20/2010
```

这种记法也可以用来引用命名分组，写作\g<*name*>。

```
regex = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})"
replacement = r"\g<month>/\g<day>/\g<year>"
print re.sub(regex, replacement, "2010-12-20")
12/20/2010
```

repl 也可以是一个函数，它接收一个 MatchObject 对象，返回一个 String 对象。这样替换功能就强大了许多，比如把所有单词统一为首字母大写格式。

```
def capitalize(match) :
    #正则表达式已经用两个捕获分组分开第一个字母和之后的字母
    return match.group(1).upper() + match.group(2).lower()
print re.sub(r"(?i)\b([a-z])([a-z]+)\b", capitalize, "one TWO tHREE ")
One TWO THREE
```

也可以设定 *count* 参数，它指定替换操作最多能发生的次数。

```
print re.sub(r"\bt[a-z]+\b", toUpper, "one two three", 1)
one TWO three
```

14.4　常用操作示例

14.4.1　验证

简单验证

```
#应当在正则表达式首尾加上\A 和\Z
re.search(r"\A\d{4}-\d{2}-\d{2}\Z", "2010-12-20") != None  # => True
```

复用 RegexObject 验证

```
dateRegex = re.compile(r"\A\d{4}-\d{2}-\d{2}\Z")
re.search(dateRegex, "2010-12-20") != None        # => True
```

14.4.2 提取

简单提取

```
regex = r"\d{4}-\d{2}-\d{2}"
string = "2010-12-20 2011-02-14"
for dateStr in re.findall(r"\d{4}-\d{2}-\d{2}", "2010-12-20 2011-02-14") :
    print dateStr
2010-12-20
2011-02-14
```

#出现捕获分组之后，返回的结果不会包含默认的捕获分组 0，所以显式加上分组 0

```
regex = r"((\d{4})-(\d{2})-(\d{2}))"
string = "2010-12-20 2011-02-14"
for matchTuple in re.findall(regex, string) :
    print "date:", matchTuple[0],
    print "year:", matchTuple[1],
    print "month:", matchTuple[2],
    print "day:", matchTuple[3]
date: 2010-12-20 year: 2010 month: 12 day: 20
date: 2011-02-14 year: 2011 month: 02 day: 14
```

复用 RegexObject 对象

```
regex = re.compile(r"((\d{4})-(\d{2})-(\d{2}))")
string = "2010-12-20 2011-02-14"
for matchTuple in regex.findall(string) :
    print "date:", matchTuple[0],
    print "year:", matchTuple[1],
    print "month:", matchTuple[2],
    print "day:", matchTuple[3]
date: 2010-12-20 year: 2010 month: 12 day: 20
date: 2011-02-14 year: 2011 month: 02 day: 14
```

使用 finditer 迭代查找

```
regex = re.compile(r"((\d{4})-(\d{2})-(\d{2}))")
string = "2010-12-20 2011-02-14"
for iter in regex.finditer(string) :
    print "date:", iter.group(0),
    print "year:", iter.group(1),
    print "month:", iter.group(2),
    print "day:", iter.group(3)
```

```
date: 2010-12-20 year: 2010 month: 12 day: 20
date: 2011-02-14 year: 2011 month: 02 day: 14
```

使用命名分组

```
regex = re.compile(r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})")
string = "2010-12-20 2011-02-14"
for iter in regex.finditer(string) :
    print "date:", iter.group(),
    print "year:", iter.group("year"),
    print "month:", iter.group("month"),
    print "day:", iter.group("day")
date: 2010-12-20 year: 2010 month: 12 day: 20
date: 2011-02-14 year: 2011 month: 02 day: 14
```

14.4.3　替换

简单替换

```
regex = r"(\d{4})-(\d{2})-(\d{2})"
replacement = r"\g<2>/\g<3>/\g<1>"
print re.sub(regex, replacement, "2010-12-20")
12/20/2010
```

命名分组替换

```
regex = r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})"
replacement = r"\g<month>/\g<day>/\g<year>"
print re.sub(regex, replacement, "2010-12-20")
12/20/2010
```

在替换中使用整个表达式匹配的文本

```
regex = r"\d+\.\d{0,2}"
replacement = r"$\g<0>"
print re.sub(regex, replacement, "the price is 12.99")
the price is $12.99
```

14.4.4　切分

```
regex = r"\s+"
for word in re.split(regex, "one\ttwo\n three") :
    print word
one
two
three
```

第 15 章　Ruby

Ruby内建了对正则表达式的支持，提供了相对完备的正则表达式功能。但Ruby 1.8 的正则表达式也存在不少限制，最突出的是不支持逆序环视。Ruby 1.9 采用全新的Oniguruma（鬼车）正则引擎，补充了包括逆序环视在内的许多功能，最重要的是全面支持Unicode（但是也造成了一些奇怪的现象，下面你会看到）。不过考虑到目前Ruby 1.9 并没有完全取代Ruby 1.8，所以本章中我们会讲解Ruby 1.8 和Ruby 1.9 的差异，本章末尾详细说明了Ruby 1.9 中正则表达式的功能变化，有兴趣的读者可以参考。[1]

15.1　预备知识

Ruby的正则表达式对象是`Regexp`。[2] 最常见的生成方法类似`/regex/`，即在首尾两个斜线/之间，用正则文字给出正则表达式，比如`/\d+/`。

为继续讲解方便，先介绍进行正则表达式匹配的方法：Ruby 中的正则匹配非常简单，直接使用操作符`=~`连接正则表达式与字符串即可。如果可以匹配，则返回第一次匹配发生的起始位置，否则返回 nil。请注意，正则表达式和字符串的左右顺序并不影响结果。

```
"0" =~ /\d/ # =>0
"a" =~ /\d/ # =>nil
/\d/ =~ "0" # =>0
/\d/ =~ "a" # =>nil
```

Ruby中有两种字符串：双引号字符串和单引号字符串。两者的主要区别在于：双引号字符串需要进行字符串转义处理，而单引号字符串不需要进行字符串转义处理。[3] 所以如果要作为正则

---

[1] 本书中所说的 Ruby 1.9 指的是 Ruby 1.9.1 及以上版本，而不包括 Ruby 1.9.0，因为这个版本用的人极少，而且经过笔者测试，Ruby 1.9.0 的正则表达式在不同操作系统中的表现差异非常大。如果你使用的是 Ubuntu 之类的操作系统，通过 apt-get 安装的 Ruby 1.9 很可能是 Ruby 1.9.0，请注意这一点。

[2] 如果你用过 Ruby 中的正则表达式，可能会发现 Ruby 和 JavaScript 中的正则表达式对象名字很像，唯一的区别在于 Ruby 的 Regexp 中 e 是小写，而 JavaScript 的 RegExp 中 E 是大写。

[3] 关于字符串和转义的详细解释，请参考本书第 6 章。

表达式的字符串，最好使用单引号字符串，这样就可以直接使用\d，而不用写成\\d；作为普通文本的字符串，则最好使用双引号字符串。

15.2　正则功能详解

15.2.1　列表

表 15-1 列出了 Ruby 的正则功能。

表 15–1　Ruby 的正则功能

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	支持 Unicode
POSIX 字符组 ☞15	`[:digit:]`	1.8 只能匹配 ASCII 字符，1.9 可以匹配 Unicode 字符
Unicode 属性 ☞127	`\p{…}`	1.9 支持
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	采用 ASCII 匹配规则
行起始位置 ☞62	`^` `\A`	默认采用多行模式
行结束位置 ☞62	`$` `\z` `\Z`	默认采用多行模式
单词边界 ☞123	`\b` `\B`	1.8 采用 ASCII 匹配规则，1.9 采用 Unicode 匹配规则
顺序环视 ☞69	`(?=…)` `(?!…)`	支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	1.9 支持
匹配模式 ☞83	**i m o x u**	支持
模式作用范围 ☞91	`(?`*modifier*`)` `(?-`*modifier*`)`	支持
纯文本模式 ☞101	`\Q…\E`	不支持
捕获分组及引用☞44	`(…)` `\`*num* **$***num*	支持
命名分组 ☞53	`(?<`*name*`>…)` `\k<`*name*`>`	1.9 支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	1.9 支持

15.2.2　字符组

Ruby 中的正则引擎可以正确识别 ASCII 编码，但无法正确识别 GBK 编码，因此无法避免 GBK 编码的“错误匹配”问题（☞116）。

```
"遭" =~ /[正则]/ # => 0
```

解决的办法是使用 Unicode 编码环境，指定程序使用 Unicode 编码的办法是在源文件的顶部写上这样一行：

```
# encoding:UTF-8
```

Ruby 1.9 之后版本的正则表达式才提供了相对完备的 Unicode 支持，如果你使用的是 Ruby 1.8，正则表达式中又有非 ASCII 字符，则最好在表达式末尾写上 u 显式指定 Unicode 模式（Ruby1.9 中，正则表达式默认使用 Unicode 编码，所以不用显式指定）。

```
"遭" =~ /[正则]/u   # => nil
"发" =~ /[正则]/u   # => 0
```

另外要注意的是，在 Ruby 1.8 中，即便指定程序使用 Unicode 编码，匹配位置仍然是按照字节计算的，Ruby 1.9 中则是按照字符计算的。

```
Ruby 1.8
#UTF-8 编码每个汉字占 3 字节，所以匹配的“发”从 3 开始
"中正" =~ /[正则]/u  # => 3
Ruby 1.9
"中正" =~ /[正则]/   # => 1
```

在 Ruby 中可以用字符组`[\u{4E00}-\u{9FFF}]`匹配任意中文字符，需要注意的是，这种写法只有 Ruby 1.9 以上版本才支持，并且使用时必须显式指定 Unicode 模式。

```
"我" =~ /[\u{4E00}-\u{9FFF}]/u   # => 0
"你" =~ /[\u{4E00}-\u{9FFF}]/u   # => 0
"a"  =~ /[\u{4E00}-\u{9FFF}]/u   # => nil
```

Ruby 支持 POSIX 字符组。在 Ruby 1.8 中，POSIX 字符组只能匹配 ASCII 字符（指定 Unicode 模式也不行），而且不可使用`[:ascii:]`和`[:word:]`；而在 Ruby 1.9 中，POSIX 字符组可以匹配 Unicode 字符，同时支持`[:ascii:]`和`[:word:]`。

```
Ruby 1.8
"0" =~ /[[:digit:]]/    # => 0
"a" =~ /[[:alpha:]]/    # => 0

Ruby 1.9
"0" =~ /[[:ASCII:]]/    # => 0
"a" =~ /[[:word:]]/     # => 0
"我" =~ /[[:word:]]/    # => 0
#全角数字
"０" =~ /[[:digit:]]/   # => 0
```

15.2.3 Unicode 属性

Ruby 1.9 支持使用 Unicode Script，只是使用 Unicode Script 时，必须显式指定 u 模式，否则

会报错。所以，如果要匹配中文字符，则可以使用\p{Han}。

```
"a"  =~ /\p{Han}/u      # => nil
"我" =~ /\p{Han}/u      # => 0
"我" =~ /\p{Han}/       # 报错
```

15.2.4　字符组简记法

Ruby的字符组简记法都采用ASCII编码匹配规则。如果指定了Unicode模式，则会换用Unicode匹配规则。[1]

```
#半角字符
"1"  =~ /\d/   # => 0
"a"  =~ /\w/   # => 0
" "  =~ /\s/   # => 0
#全角字符及中文字符
"１"  =~ /\d/  # => nil
"我"  =~ /\w/  # => nil
"　"  =~ /\s/  # => nil
```

15.2.5　单词边界

在 Ruby 1.8 中，如果没有指定 Unicode 模式，\w采用 ASCII 匹配规则，单词边界\b也采用这种匹配规则（☞120）。

```
#英文字符-半角标点
"a," =~ /a\b/       # => 0
#英文字符-全角标点
"a，" =~ /a\b/      # => 0
#英文字符-结束
"a" =~ /a\b/        # => 0
#中文字符-半角标点
"我," =~ /我\b/     # => nil
#中文字符-全角标点
"我，" =~ /我\b/    # => nil
#中文字符-空字符串
"我" =~ /我\b/      # => nil
#全角标点-结束
"，" =~ /，\b/      # => nil
#中文字符-英文字符
"我a" =~ /我\b/     # => 0
```

[1] 如果你使用的是 Ruby 1.9.0，测试结果可能与这里的完全不同。如本章开头所述，Ruby 1.9.0 是一个非常奇怪的版本，不推荐使用。

如果指定了 Unicode 模式，则\b 也采用 Unicode 匹配规则。

```
#英文字符-半角标点
"a," =~ /a\b/u      # => 0
#英文字符-全角标点
"a，" =~ /a\b/u     # => 0
#英文字符-结束
"a" =~ /a\b/u       # => 0
#中文字符-半角标点
"我," =~ /我\b/u    # => 0
#中文字符-全角标点
"我，" =~ /我\b/u   # => 0
#中文字符-空字符串
"我" =~ /我\b/u     # => 0
#全角标点-结束
"，" =~ /，\b/u     # => 0
#中文字符-英文字符
"我a" =~ /我\b/u    # => 0
```

Ruby 1.9 中的\w 虽然采用 ASCII 匹配规则，但\b 的表现与 Ruby 1.8 中指定 Unicode 方式一样，这一点值得特别注意。

15.2.6 行起始/结束位置

Ruby 默认采用多行模式，所以^不但可以匹配整个字符串的开始位置，还可以匹配字符串内部的文本行的起始位置；而\A 无论在什么情况下都匹配整个字符串的起始位置。

```
"1\n2\n" =~ /^1/    # => 0
"1\n2\n" =~ /^2/    # => nil

"1\n2\n" =~ /\A1/   # =>2
"1\n2\n" =~ /\A2/   # => nil
```

同样，因为默认采用多行模式，所以$可以匹配文本内部行的结束位置，\Z、\z 匹配的是整个字符串的结束位置（如果结束位置有换行符，则\Z 匹配这个换行符之前的位置）。

```
"1\n2\n" =~ /1$/    # => 0
"1\n2\n" =~ /2$/    # => 2
"1\n2" =~ /2$/      # => 2

"1\n2\n" =~ /1\Z/   # => nil
"1\n2\n" =~ /2\Z/   # => 2
"1\n2" =~ /2\Z/     # => 2

"1\n2\n" =~ /2\z/   # => nil
"1\n2" =~ /2\z/     # => 2
```

重复一次，Ruby 的 m 模式不等于其他语言中的多行模式，而是等于通常的单行模式，它只影响点号.的匹配。

15.2.7　环视

在 Ruby 中的肯定顺序环视(?=…)和否定顺序环视(?!…)内可以使用任意形式的子表达式。

```
"cab" =~ /c(?=ab+)/          # => 0
"ccd" =~ /c(?=(ab+|cd))/     # => 0
"cabbbb" =~ /c(?=(ab+|cd))/ # => 0
```

对逆序环视的支持则不那么好：Ruby 1.8 完全不支持逆序环视；Ruby 1.9 支持逆序环视，但是逆序环视中的表达式匹配的文本长度必须是固定的，完全不能使用+、*、?之类的量词：如果使用了多选结构，只能出现竖线，比如 *regex1|regex2*，而不能出现括号，比如*(regex1|regex2)*。

```
"ab" =~ /ab(?<=ab)/          # => 0
"cd" =~ /cd(?<!ab|cde)/      # => 0

#ab?能匹配的文本长度不固定，报错
"ab" =~ /ab(?<=ab?)/
#多选结构(ab|cd)包含括号，报错
"ab" =~ /ab(?<=(ab|cd))/
```

15.2.8　匹配模式

表 15-2 列出了 Ruby 的正则表达式支持的匹配模式。

表 15–2　Ruby 的正则表达式支持的匹配模式

常量	修饰符	解释
Regexp::IGNORECASE	i	不区分 ASCII 字符的大小写
Regexp::EXTENDED	x	允许正则表达式中出现注释，表达式中的空白字符，以及#开始到行末的文本，都视为注释
Regexp::MULTILINE	m	等于通常所说的“单行模式”，允许点号匹配任何字符，包括换行符
	u	对字符组简记法采取 Unicode 匹配规则
	o	对包含插值（#{*variable*}）的正则表达式，只在第一次遇到时求值，以后使用时不变；否则每次遇到时都求值

注：修饰符 u 其实是用来指定正则表达式编码的，可选的值一共有 4 个：n，表示不指定；e，表示 EUC（Extended UNIX Code，一种主要用于中日韩文字的多字节编码）；u，表示 Unicode；s，表示 SJIS（Shift JIS，一种主要用于日文的多字节编码）。通常只会用到 u。

匹配模式可以在表达式中以(?*modifier*)指定，或者在结束分隔符之后写上模式对应的修饰符，也可以在生成 Regexp 对象时，通过预定义常量作为参数指定。

```
"A" =~ /a/              # => nil
"A" =~ /(?i)a/          # => 0
"A" =~ /a/i             # => 0
"A" =~ Regexp.new('a', Regexp::IGNORECASE)     # => 0
```

Ruby 支持用(?-*modifier*)停用某个模式。

```
"aBb" =~ /a(?i)b(?-i)b/     # => 0
"aBB" =~ /a(?i)b(?-i)b/     # => nil
```

15.2.9 捕获分组的引用

在 Ruby 中，如果要在正则表达式内部引用捕获分组，则应当使用*num* 记法，其中 *num* 为对应捕获分组的编号。

```
"ab" =~ /([a-z])\1/      # => nil
"aa" =~ /([a-z])\1/      # => 0
```

如果要在替换时引用捕获分组，则应当使用$*num* 记法。

```
"2010-12-20".sub(/(\d{4})-(\d{2})-(\d{2})/, '\2/\3/\1')
12/20/2010
```

如果使用了命名分组，在表达式内部则应当使用\k<*name*>来引用。

```
"ab" =~ /(?<char>[a-z])\k<char>/      # => nil
"aa" =~ /(?<char>[a-z])\k<char>/      # => 0
```

在 *replacement* 字符串中同样使用\k<*name*>来引用。

```
"2010-12-20".sub(/(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/,
'\k<month>/\k<day>/\k<year>')
12/20/2010
```

15.3 正则 API 简介

15.3.1 Regexp

Ruby 中的正则表达式对象是 Regexp，一般使用/*regex*/来生成。如果习惯更为“正统”的模式，也可以用 Regexp.new()来生成，参数是表达式对应的字符串。

```
Regexp.new('[a-z]') =~ "a"  # => 0
```

之前讲过，Ruby 中有两种字符串，其中双引号字符串需要进行字符串转义处理，而单引号字符串不需要进行字符串转义处理。所以，作为正则表达式的字符串，最好使用单引号字符串。

如果要指定匹配模式，可以添加第 2 个参数（虽然实际参数是一个整数，但是 Ruby 其实有 3 个预定义常量，即 Regexp::IGNORECASE、Regexp::EXTENDED、Regexp::MULTILINE）。

```
Regexp.new('[a-z]', Regexp::IGNORECASE) =~ "a"         # => 0
Regexp.new('[a-z] #comment', Regexp::EXTENDED) =~ "a"  # => 0
Regexp.new('.', Regexp::MULTILINE) =~ "\n"             # => 0
```

Ruby 中也可以混用多个模式，用|连接预定义常量。

```
Regexp.new('[a-z]#comment', Regexp::IGNORECASE|Regexp::EXTENDED) =~ 'A' # => 0
```

Regexp.new()的第 1 个参数也可以不用字符串，而是用类似/*regex*/的形式。

```
Regexp.new(/[a-z]/) =~ "a" # => 0
```

如果使用这种形式，那么有三种形式指定匹配模式，下面各举一个例子，其结果相同。

```
Regexp.new(/(?i)[a-z]/) =~ "a"  # => 0
Regexp.new(/[a-z]/i) =~ "a"     # => 0
Regexp.new(/[a-z]/, Regexp::IGNORECASE) =~ "a" # => 0
```

另外还有 Regexp.compile()方法，它与 Regexp.new()完全一致。

值得注意的还有 Regexp 对象的编码。在 Ruby 1.9 中，Regexp 对象自身是具有编码的，在默认情况下，如果正则表达式中出现的都是 ASCII 字符，且程序采用了兼容 ASCII 的编码，则正则表达式默认采用 ASCII 编码；否则，采用与程序相同的编码。调用 Regexp.encoding?可以得到编码信息。

```
/abc/.encoding          #=> #<Encoding:ASCII>

#设定了源代码使用 UTF-8 编码
/你我他/.encoding       #=> # <Encoding:UTF-8>

#也可以显式指定 Regexp 的编码
/abc/u.encoding         #=> #<Encoding:UTF-8>
```

要正确使用具有编码属性的正则表达式，只有两种情况：Regexp 与字符串使用相同的编码；或者 Regexp 使用 ASCII 编码，而字符串使用兼容 ASCII 的编码，否则会报错。

```
Regexp.new('a'.encode('iso-8859-1')) =~ "你我他"
#=> Encoding::CompatibilityError: incompatible encoding regexp match
(ISO-8859-1 regexp with UTF-8 string)
```

如果不确定使用的 Regexp 对象的编码是否兼容 ASCII，可以使用 fix_encoding?来判断。不兼容的情况下会返回 true；如果一个 Regexp 对象的编码兼容 ASCII，它可以尝试匹配任何使

用了 ASCII 编码的字符串，而不会报错。

```
Regexp.new('你我'.encode('gbk')).fixed_encoding?       # => True
Regexp.new('你我'.encode('utf-8')).fixed_encoding?     # => True
```

15.3.2 Regexp.match(text)

之前看到的匹配都是依靠操作符=~进行的，许多读者可能不熟悉，所以这里先详细介绍它。=~用来执行正则表达式和字符串的匹配，返回匹配成功的起始位置；如果匹配不成功，则返回nil。因此，可以用这个办法来进行数据验证（验证数据时，别忘了在正则表达式两端加上\A和\z）。

```
("2010-12-20"  =~ /\A\d{4}-\d{2}-\d{2}\z/)  != nil     #  => True
("2010-12" =~ /\A\d{4}-\d{2}-\d{2}\z/)  != nil         #  => False
("a2010-12-20"  =~ /\A\d{4}-\d{2}-\d{2}\z/)  != nil    #  => False
```

如果匹配成功，会生成一个 MatchData 对象，通过 Regexp.last_match 可以获得它，其中包含了匹配的详细信息。

```
"Date: 2010-12-20, "  =~ /(\d{4})-(\d{2})-(\d{2})/
#获得 MatchData 对象
data = Regexp.last_match
# => #<MatchData "2010-12-20" 1:"2010" 2:"12" 3:"20">

#正则表达式中捕获分组数目+1，因为整个表达式匹配的文本作为编号 0
data.size()    # => 4
#第 0 号分组，也就是整个表达式匹配的文本
data[0]        # => "2010-12-20"
#第 1、2、3 号分组，各捕获分组匹配的文本
data[1]        # => "2010"
data[2]        # => "12"
data[3]        # => "20"
#编号最大的捕获分组匹配的文本
data[-1]       # => "20"
#各捕获分组匹配的文本，编号从 1 开始
data.captures  # => ["2010", "12", "20"]
#编号为 1 的捕获分组的开始位置、结束位置、范围
data.begin(1)      # => 0
data.end(1)        # => 4
data.offset(1)     #[0, 4]
#原字符串中，表达式所匹配文本左侧的文本
data.pre_match     # => "Date: "
#原字符串中，表达式所匹配文本右侧的文本
data.post_match    # => ", "
```

也可以不用显式通过 MatchData 对象来获得这些信息，Ruby 提供了若干特殊变量，使用它

们同样也可以达到目的。表 15-3 列出了常用的特殊变量。

表 15–3　Ruby 提供的特殊变量

特殊变量	等价
$~	Regexp.last_match
$&	Regexp.last_match[0]
$`	Regexp.last_match.pre_match
$’	Regexp.last_match.post_match
$1	Regexp.last_match[1]
$2	Regexp.last_match[2]
……	……
$+	Regexp.last_match[-1]

Regexp.match(*text*)方法与=~差不多，主要的区别在于，它返回的并不是匹配的起始位置，而是一个 MatchData 对象，如果匹配不成功，则返回 nil。所以，上面的代码可以这样获得 MatchData。

```
data = /\A(\d{4})-(\d{2})-(\d{2})\z/.match("2010-12-20")
```

Regex.match(*string*)与=~的另一点不同是，它可以通过可选参数设定偏移值，设定尝试匹配的开始位置，这样就可以在字符串中逐次找到匹配。在下面的例子中，每次匹配成功之后，记录下匹配的结束位置，作为下次开始尝试的起点，就实现了逐次匹配。

```
regex = /\d{4}-\d{2}-\d{2}/
text = "2010-12-20 2011-02-14"
data = regex.match(text)
while data != nil
 puts data[0]
 data = regex.match(text, data.end(0))
end
2010-12-20
2011-02-14
```

15.3.3　Regexp.quote(text)和 Regexp.escape(text)

这两个方法是等价的，它们消除 *text* 中可能出现的任何元字符的特殊含义。

```
Regexp.new(Regexp.quote("[a-z]")) =~ "[a-z]"  # => 0
Regexp.new(Regexp.quote("[a-z]")) =~ "a"      # => nil
Regexp.new(Regexp.quote("[a-z]")) =~ "z"      # => nil

Regexp.new(Regexp.escape("[a-z]")) =~ "[a-z]" # => 0
Regexp.new(Regexp.escape("[a-z]")) =~ "a"     # => nil
Regexp.new(Regexp.escape("[a-z]")) =~ "z"     # => nil
```

15.3.4 String.index(Regexp)

这个方法默认等价于=~操作，如果匹配成功，则返回匹配开始的位置，否则返回 nil。

如果不指定第 2 个参数，它等价于 Regex.match()或者=~。

```
"0".index(/\d/)    # => 0
"a0".index(/\d/)   # => 1
"a".index(/\d/)    # => nil
```

它也可以指定第 2 个参数，设定开始尝试匹配的偏移值。此时可以遍历文本，获得逐次匹配的结果。

```
regex = /\d{4}-\d{2}-\d{2}/
text = "2010-12-20 2011-02-14"
offset = text.index(regex)
while offset != nil
 puts Regexp.last_match[0]
 offset = text.index(regex, Regexp.last_match.end(0))
end
```

```
2010-12-20
2011-02-14
```

15.3.5 String.scan(Regexp)

=~操作符和 Regexp.match(*string*)方法只能进行单次匹配，如果希望一次性找到某个正则表达式在字符串中的所有匹配，就应该使用 String.scan(*Regexp*)，它返回各次匹配所对应文本组成的数组。

```
puts "2010-12-20 2011-02-14".scan(/\d{4}-\d{2}-\d{2}/)
["2010-12-20",  "2011-02-14"]
```

如果正则表达式包含捕获分组，则返回数组的元素本身也是数组。

```
puts "2010-12-20 2011-02-14".scan(/(\d{4})-(\d{2})-(\d{2})/)
[["2010", "12", "20"], ["2011", "02", "14"]]
```

请注意，这时候并不会返回整个表达式匹配的文本（也就是编号为 0 的捕获分组捕获的文本），如果需要得到整个表达式匹配的文本，可以给整个表达式首尾加上括号。

```
puts "2010-12-20 2011-02-14".scan(/((\d{4})-(\d{2})-(\d{2}))/)
[["2010-12-20", "2010", "12", "20"], ["2011-02-14", "2011", "02", "14"]]
```

也可以用 String.scan()进行更复杂的字符串处理，方法是在括号后面加上{|*match*, …|*block*}，其中 *match*,…表示各捕获分组匹配的文本，而 *block* 则是对这些文本进行处理的程序代

码，比如下面这样。

```
#匹配同时将单词转为大写输出
"one two three".scan(/\b\w+\b/) {|x| puts x.upcase}
ONE
TWO
THREE
one two three
```

15.3.6　String.slice(Regexp)

这个方法返回 Regexp 在 *String* 中能匹配的文本。

```
"a0".slice(/\d/) # => "0"
```

也可以设定第 2 个参数，指定对应捕获分组的编号，返回对应的捕获分组匹配的文本。

```
string = "Date: 2010-12-20"
regex = /(\d{4})-(\d{2})-(\d{2})/
string.slice(regex, 1) # => "2010"
string.slice(regex, 2) # => "12"
string.slice(regex, 3) # => "20"
```

如果正则表达式可以在文本中找到多次匹配，那么 String.slice(*Regexp*)只返回第一次匹配的结果。

```
string = "Date: 2010-12-20"
regex = /\d{4}-\d{2}-\d{2}/
string.slice(regex)     # => 2010-12-20
```

如果将 String.slice(*Regexp*)写成 String.slice!(*Regexp*)，则 String 自身也受到影响：在返回匹配文本的同时，还会从原字符串中删除匹配的结果。

```
text = "2010-12-20 2011-02-14"
text.slice!(/\d{4}-\d{2}-\d{2}/)
puts text
2011-02-14
```

15.3.7　String.split(Regexp)

这个方法用正则表达式来切分字符串，它返回一个数组，其中的元素是切分之后的文本。

```
"2010-12-20".split(/-/)
["2010", "12", "20"]
```

如果切分的末尾留下空字符串，在默认情况下会忽略它，而字符串开头的空字符串不会忽略。

```
"-2010-12-20-".split(/-/)
["", "2010", "12", "20"]
```

这个方法也可以指定第 2 个参数，它是一个整数，假设为 *n*，按照 *n* 取值的不同，有几种情况，以下分别举例说明。

```
# n<0，切分尽量多次，同时保留最后的空字符串；
"-2010-12-20-".split(/-/, -1)
["", "2010", "12", "20", ""]

# n=0，等于未设定此参数，切分尽量多次，不保留最后的空字符串；
"-2010-12-20-".split(/-/, 0)
["", "2010", "12", "20", ""]

# n<=limit，等于未设定此参数，切分尽量多次，不保留最后的空字符串；
"-2010-12-20-".split(/-/, 2)
["", "2010-12-20-"]

# n >limit，进行 n 次切分，末尾的空字符串会保留。
"-2010-12-20-".split(/-/, 10)
["", "2010", "12", "20", ""]
```

15.3.8 String.sub(Regexp, Str)

String.sub(*Regexp*, *Str*)用来执行正则表达式替换，*Regexp* 是匹配想要替换文本的表达式，而 *Str* 是希望更新的字符串，调用之后只会对 *Regexp* 第一次能匹配的文本进行替换。

```
puts "2010-12-20 2011-02-14".sub(/\d{4}-\d{2}-\d{2}/, 'date')
date 2011-02-14
```

如果第 1 个参数不是 *Regexp* 而是普通字符串，则进行普通字符串替换。

```
puts ".* 2011-02-14".sub('.*', 'date')
date 2011-02-14
```

也可以在 *String* 参数中，用*num* 引用 *Regexp* 对应捕获分组所匹配的文本。

```
puts "2010-12-20".sub(/(\d{4})-(\d{2})-(\d{2})/, '\2/\3/\1')
12/20/2010
```

如果需要进行更灵活的替换，则可以使用 String.sub(Regexp){| *match* | *block*}方式，其中的 *match* 是 *Regexp* 所匹配的文本，即一个 String 对象，我们可以写一段代码作为 *block*，进行更复杂的处理，比如将单词转换为大写。

```
#注意只会转换处理第一次匹配的文本
puts "one two three".sub(/[a-z]+/){| s | s.upcase}
ONE two three
```

如果需要在 *block* 中获得其他匹配的内容，可以使用之前讲过的$1、$2、$`等特殊变量。

```
regex = /(\d{4})-(\d{2})-(\d{2})/
puts "2010-12-20".sub(regex){|s| $2 + "/" + $3 + "/" + $1}
12/20/2010
```

String.sub(*Regexp*, *String*)方法并不会修改原来的字符串，它会先复制原字符串，对其进行修改之后返回。如果 *String* 对象包含的内容很多，复制可能比较消耗资源，因此 Ruby 也提供了直接对原字符串生效的方法 String.sub!(*Regexp*, *String*)。

```
string = "2010-12-20 2011-02-14"
string.sub(/\d{4}-\d{2}-\d{2}/, 'date')
puts string
date 2011-02-14
```

如果表达式在 String 中能多次匹配，String.sub()只会对第一次匹配进行替换；如果需要对所有匹配都进行替换，则需要使用 String.gsub(*Regexp*, *String*)。下面来看这个方法。

15.3.9　String.gsub(Regexp, String)

这个方法在使用上完全等价于 String.sub(*Regexp*, *String*)，唯一不同的是它会对每次匹配都进行替换。

```
puts "2010-12-20 2011-02-14".gsub(/\d{4}-\d{2}-\d{2}/, '\2/\3/\1')
12/20/2010 02/14/2011
```

相应的，String.gsub!(*Regexp*, *String*)也会修改原来的字符串。

```
string = "2010-12-20 2011-02-14"
string.gsub!(/\d{4}-\d{2}-\d{2}/, '\2/\3/\1')
puts string
12/20/2010 02/14/2011
```

15.4　常用操作示例

15.4.1　验证

简单验证

```
#不要忘记在正则表达式首尾加上\A 和\z
("2010-12-20"  =~ /\A\d{4}-\d{2}-\d{2}\z/)  != nil     #=> True

#也可以使用 String.index(Regexp)
#同样要在正则表达式首尾加上\A 和\z
"2010-12-20".index(/\A\d{4}-\d{2}-\d{2}\z/)  != nil    #=> True
```

15.4.2 提取

逐步提取

```
regex = /(\d{4})-(\d{2})-(\d{2})/
text = "2010-12-20 2011-02-14"
data = regex.match(text)
while data != nil
 print "date:", data[0], " year:", data[1]," month:", data[2], " day:", data[3], "\n"
 data = regex.match(text, data.end(0))
end
```

```
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

一次性提取

```
regex = /((\d{4})-(\d{2})-(\d{2}))/
text = "2010-12-20 2011-02-14"
all = text.scan(regex)
for one in all do
    print "date:", one[0], " year:", one[1]," month:", one[2], " day:", one[3], "\n"
end
```

```
date: 2010-12-20, year: 2010, month: 12, day: 20
date: 2011-02-14, year: 2011, month: 02, day: 14
```

15.4.3 替换

简单替换

```
puts "2010-12-20 2011-02-14".gsub(/(\d{4})-(\d{2})-(\d{2})/, '\2/\3/\1')
```

```
12/20/2010 02/14/2011
```

在 replacement 中使用整个表达式匹配的文本

```
#必须给整个表达式添加捕获型括号
regex = /(\d+\.\d{0,2})/
replacement = '$\0'
"the price is 12.99".sub(regex, replacement)
```

```
the price is $12.99
```

15.4.4 切分

```
for word in "one\ttwo\n three".split(/\s+/) do
    puts word
end
```

```
one two three
```

15.5　Ruby 1.9 的新变化

Ruby 1.8 的正则表达式还不够完善，许多常用的特性不能支持，而 Ruby 1.9 采用了全新的 Oniguruma（鬼车）引擎，对正则表达式的支持则强大了很多。下面列出 Ruby 1.9 中正则表达式的主要变化，更详细的信息请参考 http://oniguruma.rubyforge.org/oniguruma/。

新增了合并多个表达式的方法

`Regexp.union(`*Regexp*，*Regexp*，…`)`

这个方法返回的仍然是一个 `Regexp`，它“合拢”了参数中所有的 `Regexp`，任何一段文本，只要能被其中某个表达式匹配，就能被 `Regexp` 匹配。

```
"a" =~ Regexp.union(/^\d+$/, /^[a-z]+$/) # => 0
"0" =~ Regexp.union(/^\d+$/, /^[a-z]+$/) # => 0
```

关于这个方法，有一个常见的误区：在数据验证时，想当然地将各条件对应的表达式用 `Regexp.union()`“归并”，但如果这些条件必须同时满足，就很可能出错。

比如需要验证的字符串有这样两条要求：第一，全部由数字或字母组成；第二，长度在 6~12 之间。如果想当然地用 `Regexp.union()`，就会出错。

```
"0123" =~ Regexp.union(/^[a-zA-Z0-9]*$/, /^.{6,12}$/) # => 0
```

这是因为 `Regexp.union()`“归并”的表达式并不要求“同时生效”，只要其中有一条能匹配，则匹配成功。此处字符串 `0123` 可以由表达式`^[a-zA-Z0-9]*$`匹配，所以返回结果 `0`。

新增了逆序环视

```
"ab" =~ /ab(?<=ab)/ # => 0
"cd" =~ /cd(?<!ab|cde)/ # => 0
```

新增了忽略优先量词

```
puts /".*?"/.match('"string 1" and "string 2"')
"string 1"
puts /".*"/.match('"string 1" and "string 2"')
"string 1" and "string 2"
```

新增了字符组中的集合运算操作符&&

```
"a" =~ /[a-z&&[^aeiou]]/ # => nil
"b" =~ /[a-z&&[^aeiou]]/ # => 0
```

新增了指定 Unicode 码值的表示法

如果指定了多字节编码，十进制和十六进制的数字序列可以用来表示单个多字节字符，因此

可以在字符组中指定多字节字符的范围。

```
"你" =~ /[\u{4E00}-\u{9FFF}]/u # => 0
"a"  =~ /[\u{4E00}-\u{9FFF}]/u # => nil
```

如果不知道自己使用的 Ruby 的版本，或者不确认其使用了 Oniguruma 引擎，则可以使用下面的函数来判断。

```
def oniguruma?
  return true if RUBY_VERSION >= "1.9.0"

  if defined?(Regexp::ENGINE)  # Is ENGINE defined
    if Regexp::ENGINE.include?('Oniguruma')
      return true             # Some version of Oniguruma
    else
      return false            # Pre-Oniguruma engine
    end
  end
```

如果希望无缝兼容 Ruby 1.8 和 1.9，不妨自己包装一个方法，根据要处理的字符串来生成对应编码的 Regexp。

```
def get_regex(pattern, str, options=0)
  if str.methods.include?(:encoding)  # Ruby 1.8 compatibility
    return Regexp.new(pattern.encode(encoding),options)
  else
    return Regexp.new(pattern, options)
  end
end
```

第 16 章　Objective-C

随着移动互联网的发展，Mac OS 和 iOS 平台在开发中占的比例越来越大。相应的，在 Mac OS 和 iOS 平台上用正则表达式处理文本的需求也越来越迫切，所以本章主要讲解在 Objective-C 下使用正则表达式进行文本处理的相关知识。掌握这些知识之后，无论是 Mac OS 还是 iOS 平台，都可以轻松驾驭。因为 Objective-C 和 Swift 是完全不同的两种语言，而截至本章写作时止（2018 年 1 月），Swift 尚未提供原生的正则表达式处理功能，只能照搬 Objective-C 的库，所以本章集中讲解 Objective-C。

Objective-C 中的字符串（而不是 C 字符串，所以前面有个@符号）直接使用 Unicode 字符，所以，Objective-C 的正则表达式对 Unicode 的兼容也非常好。

16.1　预备知识

Objective-C 中与正则表达式相关的、最重要的类是 `NSRegularExpression`，根据 Apple 的官方文档，在 iOS 4.0+、Mac OS 10.7+、tvOS 9.0+、watchOS 2.0+中都可以直接使用它。考虑到本章的写作时间（2017 年 12 月），我们可以认为，目前在所有的 Mac OS 和 iOS 设备上都已经可以直接使用 `NSRegularExpression` 了。

Objective-C 语言中的正则表达式都是以字符串的形式给出的，所以必须考虑字符串转义和正则表达式转义。如果正则表达式中出现了反斜线`\`，在通过字符串给出时，必须进行字符串转义，写为`\\`。比如正则表达式中的`\b`，在字符串中应该写作`\\b`，`\Q…\E` 则应当写作`\\Q…\\E`。稍微麻烦一点的是，如果要在正则表达式中使用反斜线`\`，它本身应当转义为`\\`，而在字符串中，两个反斜线应当分别转义，所以在字符串中应该写作`\\\\`。

```
//这个字符串只包含反斜线字符\
NSLog(@"\\\\");    // => \
```

使用 `NSRegularExpression` 不需要做任何特殊导入。参考下面的示例代码，你应该很容易了解 `NSRegularExpression` 的使用方法。

```
NSString *text = @"there are 21 boys";
```

```
NSRange searchedRange = NSMakeRange(0, [text length]);
NSError *error = nil;

NSString *pattern = @"\\d+";
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}
NSUInteger numberOfMatches = [regex numberOfMatchesInString:
     text options:0 range: searchedRange];
NSLog(@"%ld", numberOfMatches);
2
```

其中有两点需要注意：第一，构造正则表达式使用 NSRegularExpression 这个类中的 *regularExpressionWithPattern* 方法；第二，正则表达式匹配使用了 NSRegularExpression 这个类中的 *numberOfMatchesInString* 方法，它返回能成功匹配的次数。如果要得到匹配的结果，可以使用 *matchesInString* 方法。

```
NSString *text = @"2017-12-09 2014-04-03";
NSString *pattern = @"(\\d{4})-(\\d{2})-(\\d{2})";

NSRange searchedRange = NSMakeRange(0, [text length]);
NSError *error = nil;

NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}

NSArray *matches = [regex matchesInString:
     text options:0 range: searchedRange];
for (NSTextCheckingResult *match in matches) {
     NSString *matchText = [text substringWithRange:[match range]];
     NSLog(@"match: %@", matchText);
     NSRange group1 = [match rangeAtIndex:1];
     NSRange group2 = [match rangeAtIndex:2];
     NSRange group3 = [match rangeAtIndex:3];

     NSLog(@"group1: %@", [text substringWithRange:group1]);
     NSLog(@"group2: %@", [text substringWithRange:group2]);
     NSLog(@"group3: %@", [text substringWithRange:group3]);
}
match: 2017-12-09
```

```
group1: 2017
group2: 12
group3: 09
match: 2014-04-03
group1: 2014
group2: 04
group3: 03
```

考虑到 Objective-C 中进行一次正则匹配的代码比较多，为节省篇幅，后文中尽量使用类似下面的表格加以讲解。表格中的文本和正则表达式均未考虑转义，在编写程序源代码时需要注意。

Text	Regex	匹配检测
"21"	\d\d	*True*

Text	Regex	匹配文本
"21"	\d\d	21

16.2　正则功能详解

16.2.1　列表

表 16-1 列出了 Objective-C 中的正则功能。

表 16–1　Objective–C 中的正则功能列表

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	完全支持 Unicode
POSIX 字符组 ☞15	`[:digit:]`	不支持
Unicode 属性 ☞127	`\p{…}`	完全支持 Unicode
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	采用 Unicode 匹配规则
行起始位置 ☞62	`^` `\A`	支持
行结束位置 ☞62	`$` `\z` `\Z`	支持
单词边界 ☞123	`\b` `\B`	采用 Unicode 规则，可指定 ASCII
顺序环视 ☞69	`(?=…)` `(?!…)`	支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	不支持无法确定长度范围的表达式
匹配模式 ☞83	`i m s x`	支持
模式作用范围 ☞91	`(?`*`modifier`*`)` `(?-`*`modifier`*`)`	支持
纯文本模式 ☞101	`\Q…\E`	支持

（续表）

功能	记法	说明
捕获分组及引用☞44	`(…)` `\num` `$num`	支持
命名分组 ☞53	`(?<name>…)`	到 iOS 11 才提供支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

16.2.2 字符组

在 Objective-C 中，字符组中可以直接使用中文，不会出现多字节字符错误匹配的问题，点号`.`可以匹配中文字符。

Text	Regex	匹配检测
"我"	.	*True*
"遭"	[正则]	*False*

虽然文档没有说明，但代码测试表明，Objective-C 中的字符组也可以进行集合运算，通常主要用于字符组的“减”运算——如果需要匹配英文中所有小写的辅音字母，也就是从 26 个小写字母中“减去”5 个元音字母，可以写作`[[a-z]&&[^aeiou]]`。它的意思是，“从 a 到 z 的所有字符（也就是小写英文字符）”与“除 a、e、i、o、u 之外的任何字符”取交集，也就是所有的小写辅音字母。

Text	Regex	匹配检测
"a"	[[a-z]&&[^aeiou]]	*True*
"b"	[[a-z]&&[^aeiou]]	*False*

相对于`&&`的“交”运算，Objective-C 也支持在字符组内用`||`进行“并”运算，不过因为“并集”的意思就是“给字符组内部添加字符”，所以并不需要设定特殊的运算符，比如`[[0-4]||[6-9]]`就等价于`[[0-4][6-9]]`，也等价于`[0-46-9]`。虽然这几种写法都没错，但最后那种写法显然更简单，所以我推荐最后一种写法。

Text	Regex	匹配检测
"3"	[[0-4][6-9]]	*True*
"5"	[[a-z]&&[^aeiou]]	*False*

在Objective-C中可以用`\u`*hex*指定Unicode码值，其中*hex*为4位十六进制数，所以在Unicode编码环境下可以用字符组`[\u4E00-\u9FFF]`匹配所有的中文字符。因为 Objective-C 中的正则

表达式用字符串给出，所以源代码中的文本既可以写为`[\u4E00-\u9FFF]`，也可以写为`[\\u4E00-\\u9FFF]`，匹配结果并不受影响。为了保持阅读方便，建议采用前一种写法，即`[\u4E00-\u9FFF]`。

Text	Regex	匹配检测
"我"	`[\u4E00-\u9FFF]`	*True*
"A"	`[\u4E00-\u9FFF]`	*False*

因为`\u`*hex* 只能支持 4 位十六进制数，对 BMP 之外的其他 Unicode，比如 Emoji，因为码值无法用 4 位十六进制数表示，所以只能用`\U`*hex* 表示法，其中 *hex* 为 8 位十六进制数。比如😄的码值为 `U+1F601`，就应当写作`\U0001F601`。

Objective-C 中不能使用 POSIX 字符组`[:`*name*`:]`。

16.2.3　Unicode 属性

Objective-C 对 Unicode 字符组的支持比较好，它支持 Unicode Property。这样，用`\p{N}`就可以匹配所有的数字字符，包括中文的全角数字字符，比如 0 、 1 、 2 ；`\p{P}`可以匹配各种标点符号，包括中文的冒号：等。

Text	Regex	匹配检测
"１"	`\p{N}`	*True（全角数字）*
"1"	`\p{N}`	*True*
"："	`\p{P}`	*True（全角冒号）*
": "	`\p{P}`	*True*

虽然官方文档没有说明，但实际代码测试表明，Objective-C 也支持 Unicode Script。

Text	Regex	匹配检测
"ジ"	`\p{Katakana}`	*True*
"我"	`\p{Han}`	*True*

按照官方文档说明，还可以在正则表达式中使用`\N{`*UNICODE CHARACTER NAME*`}`，但是我没有找到对应的示例代码，能找到的资料均显示“用法未知”，所以不做讲解。如果哪位读者清楚详细情况，欢迎告知我。

关于 Unicode 属性的细节，请参考第 7 章。

16.2.4　字符组简记法

在 Objective-C 的正则表达式中，`\d`、`\D`、`\w`、`\W`、`\s`、`\S` 都使用 Unicode 匹配规则。也

就是说，`\d`不等价于`[0-9]`，`\w`不等价于`[0-9a-zA-Z_]`，`\s`也可以匹配 ASCII 编码之外的空白字符。这些都需要在使用中注意。

Text	Regex	匹配检测
"1"	\d	*True*
"１"	\d	*True（全角数字）*
"a"	\w	*True*
"我"	\w	*True*
" "（半角空格）	\s	*True*
"　"（全角空格）	\s	*True*

Text	Regex	匹配检测
"１"	\D	*False（全角数字）*
"我"	\W	*False*
"　"（全角空格）	\S	*False*

16.2.5 单词边界

在 Objective-C 中，默认情况下，正则表达式的单词边界`\b`的匹配保持了和`\w`同样的规则，都遵循 Unicode 匹配规则。所以在处理中英文混排的文本时要特别注意，不要想当然地用`\b`来进行英文单词边界的判断，否则很可能发生错误。

Text	Regex	匹配检测
"a，"	a\b	*True*
"a，"	a\b	*True（全角标点）*
"中文 english 混排"	\benglish\b	*False*
"中文-english-带分隔"	\benglish\b	*True*

如果这套规则不满足需求，更希望采用ASCII匹配规则，可以在正则表达式中显式指定匹配模式 [1]，也就是在正则表达式开头加上`(?w)`。请注意，`(?w)`**只影响单词边界的判断**，不影响字符组简记法`\w`、`\d`等的判断。

[1] 此处 Apple 官方文档似乎有误。在我的计算机中测试（Xcode 9.2），默认情况下即是按照 Unicode 规则匹配的，显式指定了\w 之后才采用 ASCII 规则，与官方文档的描述相反。所以本章根据实际测试的结果撰写。

Text	Regex	匹配检测
"中文 english 混排"	\benglish\b	*False*
"中文 english 带分隔"	(?w)\benglish\b	*True*
"english 中文 mix"	\b 中文\b	*False*
"english 中文 mix"	(?w)\b 中文\b	*True*

16.2.6　行起始/结束位置

^匹配的是“行开始的位置”，在默认情况下它只能匹配“整个字符串的开始位置”，如果指定使用多行模式（Multiline Mode），^可以匹配字符串内部的文本行的开始位置；而\A 无论在什么情况下都匹配“整个字符串的开始位置”。

Text	Regex	匹配检测
"1\n2\n"	^1	*True*
"1\n2\n"	^2	*False*
"1\n2\n"	(?m)^2	*True*

Text	Regex	匹配检测
"1\n2\n"	\A1	*True*
"1\n2\n"	\A2	*False*
"1\n2\n"	(?m)\A2	*False*

在默认情况下，Objective-C 中的$、\Z、\z 匹配的是整个字符串的结束位置（如果结束位置有换行符，则$和\Z 匹配这个换行符之前的位置），多行模式只会影响$的匹配，这时候它可以匹配文本内部行的结束位置，所以进行准确验证时应当使用\z。

Text	Regex	匹配检测
"1\n"	1$	*True*
"1\n2"	1$	*False*
"1\n2"	(?m)1$	*True*

Text	Regex	匹配检测
"1\n"	1\Z	*True*
"1\n2"	1\Z	*False*
"1\n2"	(?m)1\Z	*False*

Text	Regex	匹配检测
"1"	1\z	*True*
"1\n"	1\z	*False*
"1\n"	(?m)1\z	*False*

16.2.7 环视

在 Objective-C 中的肯定顺序环视`(?=...)`和否定顺序环视`(?!...)`内可以使用任意形式的子表达式。

Text	Regex	匹配检测
"cab"	c(?=ab+)	*True*
"cab"	c(?=(ab+|cd))	*True*
"cab"	c(?=(ab+|cd))	*True*

逆序环视中也可以使用子表达式，但子表达式**必须有明确长度上限**。也就是说，`(?<=ab)`、`(?<=ab?)`、`(?<=ab{0,4})`都是合法的，`(?<=ab+)`却会报错。这个问题值得注意，因为在代码编译阶段不会触发这个错误，运行时才会出现 `NSError`。

Text	Regex	匹配检测
"ab"	ab(?<=ab)	*True*
"ab"	ab+(?<=(ab|cd))	*True*
"ab"	ab+(?<=(ab{0,4}))	*True*
"ab"	ab+(?<=(ab+))	*运行时报错*

16.2.8 匹配模式

表 16-2 列出了 Objective-C 中的正则表达式支持的匹配模式，所有模式对应的常量都包含在 `NSRegularExpressionOptions` 中。

表 16–2 Objective–C 中的正则表达式支持的匹配模式

常量	修饰符	说明
NSRegularExpressionCaseInsensitive	i	不区分 ASCII 字符的大小写
NSRegularExpressionAllowCommentsAndWhitespace	x	允许正则表达式中出现注释，表达式中的空白字符，以及#开始到行末的文本，都视为注释
NSRegularExpressionIgnoreMetacharacters		把整个正则表达式视作普通字符串，取消其中所有字符的特殊含义

（续表）

常量	修饰符	说明
NSRegularExpressionAnchorsMatchLines	m	允许^和$不仅仅匹配字符串的起始和结束位置，还可以匹配字符串内部文本行的起始和结束位置
NSRegularExpressionDotMatchesLineSeparators	s	允许点号.匹配任何字符，包括换行符
NSRegularExpressionUseUnicodeWordBoundaries	w[1]	不按照 Unicode 规则识别单词边界，局限于 ASCII 字符，默认未开启
NSRegularExpressionUseUnixLineSeparators		限定.、^、$能识别的行终结符只有换行符\n，忽略\r\n 和 Unicode 行终结符等其他字符

匹配模式可以在表达式中以(?*modifier*)指定，也可以通过预定义常量指定。

Text	Regex	匹配检测
"A"	a	*False*
"A"	(?i)a	*True*

```
NSString *pattern = @"a";
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
      pattern options:NSRegularExpressionIgnoreMetacharacters error:&error];
```

Objective-C 也支持用(?-*modifier*)停用某个匹配模式。

Text	Regex	匹配检测
"aBb"	a(?i)b(?-i)b	*True*
"aBB"	a(?i)b(?-i)b	*False*

16.2.9　纯文本模式

在\Q 和\E 之内，所有的元字符和特殊结构都失去特殊意义，只能匹配它们对应的字符本身。不过在字符串中，\Q 和\E 中的反斜线也需要经过字符串转义，应该写成\\Q 和\\E。

Text	Regex	匹配检测
"A"	\Q.\E	*True*
"."	\Q.\E	*False*

[1] Apple 官方文档说明默认是“按照传统正则表达式的规则来判断单词边界”，但是在 Xcode 9.2 下反复测试，默认情况下是按照 Unicode 规则来判断单词边界的，所以本章按照实际情况来编写。

（续表）

Text	Regex	匹配检测
"[a-f]"	\Q[a-f]\E	*True*
"b"	\Q[a-f]\E	*False*

也可以通过 NSRegularExpressionOptions 来指定。

```
NSString *pattern = @"[ab]{3}";
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:NSRegularExpressionIgnoreMetacharacters error:&error];
```

16.2.10 捕获分组的引用

在 Objective-C 中，如果要在正则表达式内部引用捕获分组，应当使用`\num`记法，其中 *num* 为对应捕获分组的编号。编号必须大于等于 1，小于最大捕获分组的编号。

Text	Regex	匹配检测
"ab"	([a-z])\1	*False*
"aa"	([a-z])\1	*True*

要在替换时引用捕获分组，应当使用$*num* 记法。

```
NSString *pattern = @"(\\d{4})-(\\d{2})-(\\d{2})";
NSString *str = @"Date 2017-12-24";
NSLog(@"Before: %@",str);
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}

NSString *replaced = [regex stringByReplacingMatchesInString:
     str options:0 range:NSMakeRange(0, [str length]) withTemplate:@"$2-$3-$1"];

NSLog(@"After: %@",replaced);

Before: Date 2017-12-24
After: Date 12-24-2017
```

要在 *replacement* 字符串中表示$字符，必须使用\$转义。

```
NSString *pattern = @"\\d+\\.\\d{0,2}";
NSString *str = @"the price is 12.99";
```

```
NSLog(@"Before: %@",str);
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
      pattern options:0 error:&error];
if(error) {
      NSLog(@"%@",[error localizedDescription]);
}

NSString *replaced = [regex stringByReplacingMatchesInString:
      str options:0 range:NSMakeRange(0, [str length]) withTemplate:@"\\$$0"];

NSLog(@"After: %@",replaced);

Before: the price is 12.99
After: the price is $12.99
```

16.2.11　命名分组

在 Objective-C 中，如果要在正则表达式内部使用捕获分组，应当使用`(?<name>…)`记法，其中 *name* 为对应捕获分组的名字。

```
NSString *text = @"2017-12-09 2014-04-03";
NSString *pattern = @"(?<year>\\d{4})-(?<month>\\d{2})-(?<day>\\d{2})";

NSRange searchedRange = NSMakeRange(0, [text length]);
NSError *error = nil;

NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
      pattern options:0 error:&error];
if(error) {
      NSLog(@"%@", [error localizedDescription]);
}

regex = [regex initWithPattern:
      @"(?<year>\\d{4})-(?<day>\\d{2})-(?<month>\\d{2})" options:0 error:&error];
if(error) {
      NSLog(@"%@", [error localizedDescription]);
}

NSArray *matches = [regex matchesInString:text options:0 range: searchedRange];
for (NSTextCheckingResult *match in matches) {
      NSRange year = [match rangeWithName:@"year"];
      NSRange month = [match rangeWithName:@"month"];
      NSRange day = [match rangeWithName:@"day"];

      NSLog(@"Y:%@, M:%@, D:%@",
```

```
            [text substringWithRange:year],
            [text substringWithRange:month],
            [text substringWithRange:day]);
}

Y:2017 M:09 D:12
Y:2014 M:03 D:04
```

如果要反向引用捕获分组，应当使用`\k<name>…`记法，其中 *name* 为对应捕获分组的名字。

```
NSString *text = @"01 01";
NSString *pattern = @"(?<digits>\\d+)\\s\\k<digits>";
```

截至本章写作时（2018 年 1 月），尚未发现在替换时引用命名分组的文档，尝试通用的表示法均不可行，所以暂时认为**替换时无法引用命名分组**。如果有读者朋友知道该如何处理，请一定来信告知。

16.3 正则 API 简介

虽然本章没有讲解 Swift，在这里还是要提一下 Swift。Swift 没有原生的正则表达式的类和方法，仍然沿用 Objective-C 中的 `NSRegularExpression`，主要区别在于，标识匹配模式的常量包含在 `NSRegularExpression.Options` 中。不过考虑到实际使用时通常使用模式修饰符，这点差别可以忽略。

Objective-C 中的正则表达式在使用上与其他语言的正则表达式有所区别。首先，每次对 `NSRegularExpression` 对象调用匹配相关方法的时候，在传入目标文本时，还必须显式指定其范围。创建 `NSRegularExpression` 时有一个参数 `options`，但 Objective-C 中调用与匹配相关的方法时，也有一个参数 `options`，但两者是不同的。前者指定的是正则表达式本身的模式（不区分大小写、多行等），后者指定的是匹配时的行为（在匹配的什么阶段调用指定的代码块），后者不在本章的讲解范围内。

16.3.1 predicateWithFormat

这并不是 `NSRegularExpression` 的方法，而是 `NSPredicate` 的方法。但是，如果你只希望进行正则表达式的验证，并且需要反复验证，那么使用 `NSPredicate` 是不错的办法。你可以用正则表达式新建一个 `NSPredicate` 对象，并指定验证方法是 `MATCHES`，然后就可以调用 `evaluateWithObject` 方法，对各种正则表达式进行验证了。

```
NSString *regex = @"[0-9]{6}";
NSPredicate *tester = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", regex];
[tester evaluateWithObject:@"123456"];        // => true
```

请注意，这个方法验证的是“完整匹配”，也就是说，不必在正则表达式两端加上\A和\Z。

```
NSString *regex = @"[0-9]{6}";
NSPredicate *tester = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", regex];
[tester evaluateWithObject:@"123456"];        // => true
[tester evaluateWithObject:@"1234567"];       // => false
```

16.3.2　rangeOfString

这也不是 NSRegularExpression 的方法，而是 NSString 的方法。它返回指定特征文本在目标字符串中的位置，如果设定了 options 为 NSRegularExpressionSearch，就可以执行正则表达式搜索。

```
NSString *text = @"012345678";
NSString *regex = @"[34]+";
NSRange range = [text rangeOfString:regex options:NSRegularExpressionSearch];
NSLog(@"my range is %@", NSStringFromRange(range));

{3, 2}
```

16.3.3　regularExpressionWithPattern

Objective-C 中的正则表达式采用面向对象式处理，所以必须先创建一个 NSRegularExpression 对象，此时需要用到 regularExpressionWithPattern 这个方法。除了作为正则表达式的文本，还需要同时指定匹配模式参数 options，和用来保存可能错误的参数 error。对于匹配模式 options，推荐的做法是在正则表达式中使用模式修饰符，所以通常将其指定为 0 即可。如果模式修饰符与 options 的模式相冲突，则优先选择模式修饰符。而 error 必须事先声明，创建时传入，并且记得在创建完成之后检查它。

```
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
      @"123456" options:0 error:&error];
if(error) {
      NSLog(@"%@", [error localizedDescription]);
} else {
      //进行正则处理
}
```

16.3.4　initWithPattern

对于已经创建的 NSRegularExpression 对象，可以在保留其引用的同时重置其中的正则表达式。但是在实际开发中基本不会这样使用，所以这里不做讲解。

16.3.5 pattern

对于已经创建的 NSRegularExpression 对象，访问 pattern 可以获得文本形式的正则表达式。

```
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     @"[0-9]{6}" options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
} else {
     NSLog(@"%@", [regex pattern]);
}
```

```
[0-9]{6}
```

16.3.6 numberOfCaptureGroups

对于已经创建的 NSRegularExpression 对象，访问 numberOfCaptureGroups 可以获得其中捕获分组的数量。

```
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     @"(\\d{4})-(\\d{2})-(\\d{2})" options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
} else {
     NSLog(@"%ld", [regex numberOfCaptureGroups]);
}
```

```
3
```

16.3.7 numberOfMatchesInString

如果创建的 NSRegularExpression 对象已经进行过匹配，可以用这个方法获得成功匹配的次数。它主要有两个用途：第一，判断匹配是否成功，不过这个目的也可以用上面提到的方法来达到；第二，如果需要对各次匹配进行迭代处理，可以获得迭代次数的上限。

```
NSError *error = nil;
NSString *text = @"123456";
NSRange searchedRange = NSMakeRange(0, [text length]);
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     @"[0-9]" options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
} else {
```

```
        NSLog(@"%ld", [regex numberOfMatchesInString:
              text options:0 range:searchedRange]);
}
```

```
6
```

16.3.8　stringByReplacingMatchesInString

这个方法用来进行正则表达式替换，要替换成的文本在最后一个参数中给出，可以在其中用转义序列引用匹配的内容。

```
NSString *pattern = @"(\\d{4})-(\\d{2})-(\\d{2})";
NSString *str = @"2018-03-02";
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
      pattern options:0 error:&error];
if(error) {
      NSLog(@"%@", [error localizedDescription]);
}
NSString *replaced = [regex stringByReplacingMatchesInString:
      str options:0 range:NSMakeRange(0, [str length])
withTemplate:@"$2/$3/$1"];
NSLog(@"After: %@", replaced);
```

```
03/02/2018
```

16.3.9　replacingMatchesInString

这个方法与上面的方法类似，区别在于，它在匹配时接收的文本不是 NSString 而是 NSMutableString，调用这个方法会即时修改 NSMutableString 的内容。

```
NSString *pattern = @"(\\d{4})-(\\d{2})-(\\d{2})";
NSMutableString *str = [@"2018-03-02" mutableCopy];
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
      pattern options:0 error:&error];
if(error) {
      NSLog(@"%@", [error localizedDescription]);
}
[regex replaceMatchesInString:str options:0 range:NSMakeRange(0, [str length])
      withTemplate:@"$2/$3/$1"];
NSLog(@"After: %@", str);
```

```
03/02/2018
```

16.3.10　escapedPatternForString

这是一个类方法，去掉传入 NSString 中可能在正则表达式中有特殊含义的字符，得到“干

净”的 NSString。在处理最终用户输入的内容或从外部获得的字符串时，往往需要用到它。

```
NSString *input = @"^(N/A)$";
NSString *output = [NSRegularExpression escapedPatternForString: input];
NSLog(@"%@", output);

^\(N\/A\)\$
```

16.3.11 escapedTemplateForString

这个方法与上面的方法名字类似，所不同的是，它返回的是用作 *template* 的 NSString。这里说的 *template*，就是上面替换时类似 $2/$3/$1 的字符串。所以，其他元字符都不会变化，只有$会被转义处理。

```
NSString *input = @"^(N/A)$";
NSString *output = [NSRegularExpression escapedTemplateForString: input];
NSLog(@"%@", output);

^(N/A)\$
```

16.4 常用操作示例

16.4.1 验证

简单验证

```
//注意两端可以不加\A 和\z
NSString *regex = @"\\d{4}-\\d{2}-\\d{2}";
NSPredicate *dateValid = [NSPredicate predicateWithFormat:@"SELF MATCHES %@", regex];
[dateValid evaluateWithObject:@"2018-02-28"];      // => True
[dateValid evaluateWithObject:@"-2018-02-28-"];    // => False
```

16.4.2 提取

```
NSString *text = @"2017-12-09 2014-04-03";
NSString *pattern = @"\\d{4}-\\d{2}-\\d{2}";

NSRange searchedRange = NSMakeRange(0, [text length]);
NSError *error = nil;

NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
```

```
}

NSArray *matches = [regex matchesInString:text options:0 range: searchedRange];
for (NSTextCheckingResult *match in matches) {
     NSString *matchText = [text substringWithRange:[match range]];
     NSLog(@"%@", matchText);
}
```

```
2017-12-09
2014-04-03
```

```
NSString *text = @"2017-12-09 2014-04-03";
NSString *pattern = @"(\\d{4})-(\\d{2})-(\\d{2})";

NSRange searchedRange = NSMakeRange(0, [text length]);
NSError *error = nil;

NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}

NSArray *matches = [regex matchesInString:text options:0 range: searchedRange];
for (NSTextCheckingResult *match in matches) {
     NSString *matchText = [text substringWithRange:[match range]];
     //使用 printf 而不用 NSLog 是为了避免换行，方便观察输出
     printf("%s\t", [matchText UTF8String]);
     NSRange group1 = [match rangeAtIndex:1];
     NSRange group2 = [match rangeAtIndex:2];
     NSRange group3 = [match rangeAtIndex:3];

     printf("year: %s\t", [[text substringWithRange:group1] UTF8String]);
     printf("month: %s\t", [[text substringWithRange:group2] UTF8String]);
     printf("day: %s\n", [[text substringWithRange:group3] UTF8String]);
}
```

```
date: 2017-12-09, year: 2017, month: 12, day: 09
date: 2014-04-03, year: 2014, month: 04, day: 03
```

```
//使用命名分组
NSString *text = @"2017-12-09 2014-04-03";
NSString *pattern = @"(?<year>\\d{4})-(?<month>\\d{2})-(?<day>\\d{2})";

NSRange searchedRange = NSMakeRange(0, [text length]);
```

```
NSError *error = nil;

NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}

NSArray *matches = [regex matchesInString:text options:0 range: searchedRange];
for (NSTextCheckingResult *match in matches) {
     NSRange year = [match rangeWithName:@"year"];
     NSRange month = [match rangeWithName:@"month"];
     NSRange day = [match rangeWithName:@"day"];

     NSLog(@"Y:%@, M:%@, D:%@", [text substringWithRange:year],
          [text substringWithRange:month], [text substringWithRange:day]);
}
```

```
Y:2017 M:09 D:12
Y:2014 M:03 D:04
```

16.4.3 替换

简单替换

```
NSString *pattern = @"(\\d{4})-(\\d{2})-(\\d{2})";
NSString *str = @"Date 2017-12-24";
NSLog(@"Before: %@",str);
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}

NSString *replaced = [regex stringByReplacingMatchesInString:
     str options:0 range:NSMakeRange(0, [str length]) withTemplate:@"$2-$3-$1"];

NSLog(@"After: %@",replaced);
```

```
Before: Date 2017-12-24
After: Date 12-24-2017
```

在 replacement 中使用$

```
NSString *pattern = @"\\d+\\.\\d{0,2}";
```

```
NSString *str = @"the price is 12.99";
NSLog(@"Before: %@",str);
NSError *error = nil;
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     pattern options:0 error:&error];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}

NSString *replaced = [regex stringByReplacingMatchesInString:
     str options:0 range:NSMakeRange(0, [str length]) withTemplate:@"\\$$0"];

NSLog(@"After: %@",replaced);
```

```
Before: the price is 12.99
After: the price is $12.99
```

16.4.4　切分

简单切分（因为没有现成的切分方法，只能先替换再切分）

```
NSRegularExpression *regex = [NSRegularExpression regularExpressionWithPattern:
     @"\\s+" options:0 error:nil];
if(error) {
     NSLog(@"%@", [error localizedDescription]);
}
NSString *text = @"one two\tthree\nfour";
NSString *delimeter = @"###SPLIT###";
NSRange range = NSMakeRange(0, text.length);
if ([regex numberOfMatchesInString:text options:0 range:range] > 0) {
     NSString *body = [regex stringByReplacingMatchesInString:
          text options:0 range:range withTemplate:delimeter];
     NSArray *slices = [body componentsSeparatedByString:delimeter];
     for(NSString *slice in slices) {
          NSLog(@"%@", slice);
     }
} else {
     NSLog(@"no match");
}
```

```
one
two
three
four
```

第 17 章　Golang

Google 发明的 Go 语言（严格来说并没有“Go 语言”，所以下文简称 Golang）在近些年发展迅猛。Golang 号称“互联网时代的 C”，既保证了高效率，又提供了高级的编程范式和丰富的库，因此深受不少服务端开发人员的喜爱。与 C 和 C++不同的是，Golang 内置了正则表达式的库 `regexp`，使用正则表达式尤其方便。所以，本章主要讲解 `regexp` 的使用。

要说明的是，Golang 到目前（2018 年 1 月）来说还算“年轻”，所以并没有提供某些“业界通用”的特性，有些特性显得比较古怪（比如用某个模式修饰符交换匹配优先量词和忽略优先量词的意义）。本章的讲解基于 Golang 1.9.2，写作时我会确认所有代码都经过实际运行。如果你读到时发现同样代码的结果与描述的不符，有可能是语言特性发生了变化。

不过，Golang 的正则表达式不支持回溯的特性估计很难改变。Golang 的正则表达式基于 RE2 规范，而 RE2 的正则引擎采用的是 DFA。在第 8 章讲过，DFA“天生”不支持回溯。

同样需要注意的是，Golang 的文档虽然很详细，但是其中关于正则表达式的许多术语与业界流行的并不相同，比如多选分支不叫 alternative 而叫 composite，量词不叫 quantifier 而叫 repetitions，忽略优先量词不叫 lazy/reluctant quantifier 而叫 prefer fewer。所以如果用常见名词去搜索未必能找到答案，倘若你已经熟悉正则表达式的通行规则，阅读 Golang 的文档应当没有问题。

Golang 中的字符串直接使用 Unicode 字符，所以，Golang 的正则表达式对 Unicode 的兼容也非常好。

17.1　预备知识

Golang中与正则表达式相关的、最重要的package是`regexp`。[1] 如果要在代码中使用正则表达式，一定要在程序头部导入它（同样要注意的是，因为Golang对导入有严格限制，如果有package导入了但没有使用，编译时就会出错）。

```
import "regexp"
```

[1] Ruby、Golang、JavaScript 中的正则表达式名字很相似，唯一的区别在于大小写：Ruby 中是 Regexp，Golang 中是 regexp，JavaScript 中是 RegExp。

```
......
    matched, _ := regexp.MatchString("\\d+", "123456")
```

Golang 中也提供了“原生字符串”，也就是“特殊字符不会经过转义”的字符串，具体方法是用两个后引号（back quote，不是单引号，键盘上字符 1 左边的那个键），比如`\d`。用到正则表达式时，采用原生字符串会方便很多，如果采用普通字符串，正则表达式中的不少表示法就需要额外转义，否则会出现编译错误。

```
//编译错误！无法识别这个字符串
matched, _ := regexp.MatchString("\d+", "123456")

//编译通过
matched, _ := regexp.MatchString(`\d+`, "123456")
matched, _ := regexp.MatchString("\\d+", "123456")
```

额外的转义必然会增加出错的几率。与 Java 不同，Golang 中的正则表达式并不会处理\转义序列，所以如果要输出\，只需写为\\，如果写为\\\\则会输出\。

```
//这个字符串只包含反斜线字符\
fmt.Println("\\");      // => \
fmt.Println("\\\\");    // => \\
```

为节省篇幅，后文中关于正则表达式的讲解我尽量使用类似下面的表格加以讲解。表格中的文本和正则表达式均以源代码中的格式为准，实践时可以照原样抄入源代码中，不必考虑转义问题。

Text	Regex	匹配检测
"21"	`\d\d`	*True*

Text	Regex	匹配文本
"21"	`\d\d`	21

要注意的是，虽然 Golang 的字符串中可以直接包含 Unicode 字符，但它其实还是被当作 UTF-8 的字节数组来对待的，所以在计算字符串位置（比如截取字符串、计算表达式匹配结果的偏移值）时，得到的结果仍然是按照字节来计数的。虽然 Golang 也提供了专门针对 Unicode 字符串处理的 rune[]（其实是 uint32 数组），不幸的是，正则表达式相关的函数都不能接收 rune[] 类型的参数。所以，如果你看到正则表达式`[34]+`在字符串“012345”中匹配结果的偏移值是 3～4，而在“零壹贰 34 伍”中匹配结果的偏移值是 9~11，千万不要感到奇怪。

17.2 正则功能详解

17.2.1 列表

表 17-1 列出了 Golang 中的正则功能。

表 17–1 Golang 中的正则功能列表

功能	记法	说明
字符组 ☞2	`[…]` `[^…]`	完全支持 Unicode
POSIX 字符组 ☞15	`[:digit:]`	支持
Unicode 属性 ☞127	`\p{…}`	支持 Unicode
字符组简记法 ☞120	`\d` `\D` `\w` `\W` `\s` `\S`	采用 ASCII 匹配规则
行起始位置 ☞62	`^` `\A`	支持
行结束位置 ☞62	`$` `\z` `\Z`	支持，不支持`\Z`
单词边界 ☞123	`\b` `\B`	采用 ASCII 规则
顺序环视 ☞69	`(?=…)` `(?!…)`	不支持
逆序环视 ☞69	`(?<=…)` `(?<!…)`	不支持
匹配模式 ☞83	`i m s U`	支持
模式作用范围 ☞91	`(?modifier)` `(?-modifier)`	支持
纯文本模式 ☞101	`\Q…\E`	支持
捕获分组及引用☞44	`(…)` `\num` **$*num***	支持
命名分组 ☞53	`(?P<name>…)`	支持
非捕获分组 ☞55	`(?:…)`	支持
多选结构 ☞39	`(…|…)`	支持
匹配优先量词 ☞19	`?` `*` `+`	支持
忽略优先量词 ☞26	`??` `*?` `+?` `{n,m}?`	支持

17.2.2 字符组

Golang 中的字符串原生支持 UTF-8，所以字符组中可以直接使用中文，不会出现多字节字符错误匹配的问题，点号`.`可以匹配中文字符。

Text	Regex	匹配检测
"我"	`` `.` ``	*True*
"遭"	`` `[正则]` ``	*False*

Golang 中的字符组不支持集合运算，所以不要使用类似`[[a-z]&&[^aeiou]]`和`[[0-4]||[6-9]]`的字符组。如果出现了这样的字符组，编译时不会报错，但程序运行时的行为无法预测。

Golang 中可以用`\uhex`指定 Unicode 码值，其中 *hex* 为 4 位十六进制数，所以可以用字符组`[\u4E00-\u9FFF]`匹配所有的中文字符。要注意的是，如果使用这种表示法，就**不应当使用原生字符串**，否则`\uhex`的转义序列无法被正确识别。

Text	Regex	匹配检测
"我"	"[\u4E00-\u9FFF]"	*True*
"A"	"[\u4E00-\u9FFF]"	*False*

如果要指定 BMP 之外的 Unicode 字符，因为其码值不能用 4 位十六进制数表示，就必须使用`\Uhex`指定，其中 *hex* 为 8 位十六进制数，比如`\U0000672C`。

Golang 中可以使用 POSIX 字符组`[[:name:]]`。它们都采用 ASCII 匹配规则，请注意，必须使用两个方括号。

17.2.3　Unicode 属性

Golang 对 Unicode 字符组的支持比较好，它支持 Unicode Property（Golang 的官方文档称为 Unicode Character class）。这样，用`\p{N}`就可以匹配所有的数字字符，包括中文的全角数字字符，比如 0、1、2；`\p{P}`可以匹配各种标点符号，包括中文的冒号：等。

Text	Regex	匹配检测
"１"	`\p{N}`	*True（全角数字）*
"1"	`\p{N}`	*True*
"："	`\p{P}`	*True（全角冒号）*
":"	`\p{P}`	*True*

虽然官方文档只说明支持 Unicode Property，但实际代码测试表明，Golang 也支持 Unicode Script。

Text	Regex	匹配检测
"ジ"	`\p{Katakana}`	*True*
"我"	`\p{Han}`	*True*

关于 Unicode 属性的细节，请参考第 130 页。

17.2.4 字符组简记法

在 Golang 的正则表达式中，`\d`、`\D`、`\w`、`\W`、`\s`、`\S` 全部使用 ASCII 匹配规则。也就是说，`\d` 等价于 `[0-9]`，`\w` 等价于 `[0-9a-zA-Z_]`，`\s` 不能匹配 ASCII 编码之外的空白字符。这些都需要在使用中注意。

Text	Regex	匹配检测
"1"	`\d`	*True*
"１"	`\d`	*False（全角数字）*
"a"	`\w`	*True*
"我"	`\w`	*False*
" "	`\s`	*True*
"　"	`\s`	*False（全角空格）*

Text	Regex	匹配检测
"１"	`\D`	*True（全角数字）*
"我"	`\W`	*True*
"　"	`\S`	*True（全角空格）*

17.2.5 单词边界

在 Golang 中，默认情况下，正则表达式的单词边界 `\b` 的匹配保持了和 `\w` 同样的规则，都遵循 ASCII 匹配规则。所以处理中英文混排文本时可以放心用 `\b` 来进行英文单词边界的判断。

Text	Regex	匹配检测
"a,"	`a\b`	*True*
"a，"	`a\b`	*True（全角标点）*
"中文 english 混排"	`\benglish\b`	*True*
"中文-english-带分隔"	`\benglish\b`	*True*

17.2.6 行起始/结束位置

`^` 匹配的是“行开始的位置”，在默认情况下它只能匹配“整个字符串的开始位置”，如果指定使用多行模式（Multiline Mode），`^` 可以匹配字符串内部的文本行的开始位置；而 `\A` 无论在什么情况下都匹配“整个字符串的开始位置”。

Text	Regex	匹配检测
"1\n2\n"	`^1`	*True*
"1\n2\n"	`^2`	*False*
"1\n2\n"	`(?m)^2`	*True*

Text	Regex	匹配检测
"1\n2\n"	`\A1`	*True*
"1\n2\n"	`\A2`	*False*
"1\n2\n"	`(?m)\A2`	*False*

在默认情况下，Golang 中的$、\z匹配的是整个字符串的结束位置（与其他语言不同的是，Golang 中没有\Z，而且，如果结束位置有换行符，则$并**不会匹配**这个换行符之前的位置），多行模式只会影响到$的匹配，这时候它可以匹配文本内部的行结束位置，所以进行准确验证时应当使用\z。

Text	Regex	匹配检测
"1\n"	`1$`	*False*（请注意，与其他语言不同）
"1\n2"	`1$`	*False*
"1\n2"	`(?m)1$`	*True*

Text	Regex	匹配检测
"1"	`1\z`	*True*
"1\n"	`1\z`	*False*
"1\n"	`(?m)1\z`	*False*

17.2.7　环视

很遗憾，虽然环视已经得到广泛使用，但是到目前（1.9.2）为止，Golang 中的 regexp 并没有提供对环视的支持。如果一定要使用环视，可以采用第三方的库，或者等待 Golang 的后续版本。

17.2.8　匹配模式

表 17-2 列出了 Golang 中的正则表达式支持的常见匹配模式，所有模式对应的常量都包含在一个 uint16 的变量 *Flags* 中。

表 17-2 Golang 中的正则表达式支持的匹配模式

常量	修饰符	说明
FoldCase	i	不区分 ASCII 字符的大小写
Literal		把整个正则表达式视作普通字符串，取消其中所有字符的特殊含义
OneLine		指定^和$仅匹配字符串的起始和结束位置
	m	允许^和$不仅仅匹配字符串的起始和结束位置，还可以匹配字符串内部文本行的起始和结束位置
DotNL	s	允许点号.匹配任何字符，包括换行符
	U	交换量词意义，互换*和*?、+和+?意义（很古怪）

匹配模式可以在表达式中以(?*modifier*)指定，原理上说应当可以通过预定义常量指定，但我在网络上没有搜索到任何示范。考虑到使用修饰符本来就是业界通行的做法，我推荐大家使用模式修饰符。

Text	Regex	匹配检测
"A"	`a`	*False*
"A"	`(?i)a`	*True*

Golang 也支持用(?-*modifier*)停用某个匹配模式。

Text	Regex	匹配检测
"aBb"	`a(?i)b(?-i)b`	*True*
"aBB"	`a(?i)b(?-i)b`	*False*

17.2.9 纯文本模式

在\Q和\E之内，所有的元字符和特殊结构都失去特殊意义，只能匹配它们对应的字符本身。不过在字符串中，\Q和\E中的反斜线也需要经过字符串转义，应该写成\\Q 和\\E。

Text	Regex	匹配检测
"A"	`\Q.\E`	*True*
"."	`\Q.\E`	*False*
"[a-f]"	`\Q[a-f]\E`	*True*
"b"	`\Q[a-f]\E`	*False*

17.2.10 捕获分组的引用

在 Golang 中，如果仅使用内置的 regexp，无法在正则表达式内部引用捕获分组。如果你确

实需要用到这个特性，可以考虑第三方的库。

Text	Regex	匹配检测
"ab"	`([a-z])\1`	*False*
"aa"	`([a-z])\1`	*True*

替换时可以引用捕获分组，应当使用**$*num***记法。

```
re := regexp.MustCompile(`(\d{4})-(\d{2})-(\d{2})`)
fmt.Println(re.ReplaceAllString("2017-12-24", "$2-$3-$1"))
```

```
12-24-2017
```

要在***replacement***字符串中表示**$**字符，可以直接写**$**，不必转义。

```
re := regexp.MustCompile(`\d+`)
fmt.Println(re.ReplaceAllString("the price is 12", "$ $0"))
```

```
the price is $ 12
```

17.2.11　命名分组

在 Golang 中，如果要在正则表达式内部使用捕获分组，应当使用`(?P<name>…)`记法，其中***name***为对应捕获分组的名字。

```
var myExp = regexp.MustCompile(`(?P<first>\d+)\.(\d+).(?P<third>\d+)`)

match  := myExp.FindStringSubmatch("1234.5678.9")
result := make(map[string]string)
for i, name := range myExp.SubexpNames() {
    if i != 0 { result[name] = match[i] }
}
fmt.Printf("by name: %s %s\n", result["first"], result["third"])
```

```
by name: 1234 9
```

如果要在替换中使用命名分组，可以使用`${name}`记法。

```
var myExp = regexp.MustCompile(`(?P<first>\d+)\.(\d+).(?P<third>\d+)`)
fmt.Println(myExp.ReplaceAllString("1234.5678.9", "${first}-${third}"))
```

```
1234-9
```

截至本章写作时（2018 年 1 月），尚未发现在正则表达式内部引用命名分组的文档，尝试通用的表示法均不可行，所以暂时可以认为**正则表达式内部无法引用命名分组**。如果有读者朋友知道该如何处理，请一定来信告知。

17.3 正则 API 简介

前面说过，Golang 的正则表达式采用“面向对象”的处理方法。也就是说，在使用正则表达式的相关功能之前，必须先创建正则表达式对象。然后可以调用其他正则表达式相关的函数。比如最常见的，检查一个字符串中是否存在能由正则表达式匹配的文本。

```
package main

import (
    "fmt"
    "regexp"
)

func main() {
    re, err := regexp.Compile(`\d+`)
    if err != nil {
        fmt.Println("There is a problem with your regexp.\n")
        return
    }
    if re.MatchString("2345.") {
        fmt.Println("Match")
    }
}
```

```
Match
```

Golang 的正则表达式相关函数与其他语言中的有一个显著的区别，即它往往既提供了针对（接收参数为）string 类型的函数，也提供了针对（接收参数为）byte[]类型的函数，相应的，返回的参数类型也分别为 string 或者 byte[]。两种函数通常成对出现，有相同的处理逻辑、类似的命名。在本书中，一方面因为篇幅所限，一方面参考网络上的相关文档和教程，主要采用接收、返回参数类型均为 string 的函数，如果要用到接收、返回参数类型均为 byte[]的函数，也可以直接参考。

17.3.1 Compile 和 MustCompile

Compile 和 MustCompile 都可以用来生成正则表达式对象，两者唯一的区别是：如果正则表达式本身存在错误，MustCompile 会立刻抛出异常，所以无须考虑错误码的第二个返回值。所以，如果希望使用“创建正则表达式-调用正则表达式”的链式编程方法，MustCompile 会更加方便。

```
//不会报错
regex, err := regexp.Compile(`(a|`)
if (err != nil) {
```

```
    // do something
}

//立刻报错
regex := regexp.MustCompile(`(a|`)
```

除了 `Compile` 和 `MustCompile`，Golang 中还提供了 `CompilePOSIX` 和 `MustCompilePOSIX`。它们对应的是另一台正则引擎，即 POSIX ERE。考虑到实际情况，除非你有比较特殊的需求，否则一般不会用到 POSIX 标准的正则表达式相关特性，所以这两个函数在此不做讲解。

如果要指定匹配模式，目前**只能**在表达式中使用`(?modifier)`修饰符指定。

```
regex := regexp.MustCompile(`(?i)ab`)
```

Golang 也提供了一些静态函数，“看起来”不用编译生成正则表达式对象。但如果你仔细阅读过之前的章节就会知道，这些函数无非是一些语法糖，使用起来比较趁手，运行速度却没有什么差别。而且，如果你需要反复用到某些正则表达式，最好还是老老实实地先编译，再使用。

17.3.2　MatchString

这个函数可以检验字符串中是否存在能由正则表达式匹配的文本。要注意的是，它检验的是“字符串中（的子串）能否由表达式匹配”，所以如果你要用它来进行验证，最好在首尾加上匹配字符串起始和结束位置的锚点`\A`和`\z`。我们可以先调用 `Compile` 和 `MustCompile` 创建正则表达式对象，也可以使用静态函数直接检验。要注意的是，如果用静态函数检验，**必须同时接收返回的两个参数**，第一个参数是 bool 类型，表示能否找到。

```
regexp.MustCompile(`\d{4}`).MatchString("123456")    // => true
regexp.MustCompile(`\A\d{4}\z`).MatchString("123456")    // => false
regexp.MustCompile(`\A\d{4}\z`).MatchString("1234")    // => true

//如果使用静态函数，必须同时接收两个参数
matched, _ := regexp.MatchString((`\A\d{4}\z`, "1234"))
fmt.Println(matched)  // => true
```

17.3.3　FindString

这个函数可以提取出目标字符串中能由正则表达式匹配的、出现在最左边的文本。要注意的是，如果找不到能匹配的文本，它会返回空 `string`，如果正则表达式能匹配但匹配的“就是”空字符串，返回的仍然是空 `string`。

```
regexp.MustCompile(`\d+`).FindString("abc 1234 567")    // => 1234

regexp.MustCompile(`^`).FindString("abc")       // => 能匹配空字符串，结果为空字符串
regexp.MustCompile(`[0-9]`).FindString("abc")   // => 未匹配，结果为空字符串
```

17.3.4 FindAllString

这个函数可以视作上一个函数的完整版本，它会返回一个 string 数组，包含了目标字符串中能由正则表达式匹配的所有文本。调用这个函数时，还需要传入一个 int 参数，它指定返回数组的大小：如果 n>0，则返回的数组最多包含 n 个元素（更多匹配结果会被忽略）；如果 n=0，则返回空数组；如果 n<0，则返回所有的匹配。

```
regexp.MustCompile(`\d+`).FindAllString("12 34 567", -1)     // => ["12" "34" "567"]
regexp.MustCompile(`\d+`).FindAllString("12 34 567", 0)      // => []
regexp.MustCompile(`\d+`).FindAllString("12 34 567", 2)      // => ["12" "34"]
regexp.MustCompile(`\d+`).FindAllString("12 34 567", 6)      // => ["12" "34" "567"]
```

17.3.5 FindStringIndex

如果在目标字符串里找到了正则表达式能匹配的文本，有时希望知道这段文本在目标字符串中的位置，此时可以使用这个函数。它返回一个数组，其中包含两个 int，分别是匹配文本在目标字符串中的起始、结束偏移值。如果没有找到，返回的是空数组。

```
regexp.MustCompile(`\d+`).FindStringIndex("abc1234efg")    // => [3 7]
regexp.MustCompile(`\d+`).FindStringIndex("abcefg")        // => []
```

要补充说明的是，虽然 Golang 中的字符串直接支持 Unicode 字符，但是计算偏移值时，仍然是按照字节而不是字符来算的。如果采用 UTF-8 编码格式，则每个中文字符需要 3 字节。所以，有些时候程序的结果可能出乎意料。

```
regexp.MustCompile(`[34]+`).FindStringIndex("1234567")      // => [2 4]
regexp.MustCompile(`[34]+`).FindStringIndex("零壹贰 34567") // => [9 11]
```

17.3.6 FindAllStringIndex

这个函数也可以视作上一个函数的完整版本，不同的是它返回一个二维数组，其中的每个元素也是一个包含两个 int 的数组，分别对应每段匹配文本在目标字符串中的起始、结束偏移值。它同样需要传入一个 int 参数，指定返回数组的大小：如果 n>0，则返回的数组最多包含 n 个元素（更多匹配结果会被忽略）；如果 n=0，则返回空数组；如果 n<0，则返回所有的匹配。

```
regexp.MustCompile(`\d+`).FindAllStringIndex("12 34 567", -1)   // => [[0 2] [3 5] [6 9]]
regexp.MustCompile(`\d+`).FindAllStringIndex("12 34 567", 0)   // =>[]
regexp.MustCompile(`\d+`).FindAllStringIndex("12 34 567", 2)   // => [[0 2] [3 5]]
regexp.MustCompile(`\d+`).FindAllStringIndex("12 34 567", 4)   // => [[0 2] [3 5] [6 9]]
```

17.3.7 FindStringSubmatch

如果指定的正则表达式里存在捕获分组，这个函数可以找到（目标字符串中最左边的匹配结

果中）各个捕获分组匹配的文本。在返回的数组中，第 0 个元素为整个正则表达式匹配的文本，第 1 个元素为编号为 1 的捕获分组匹配的文本……如果没有匹配成功，则返回空数组。

```
re := regexp.MustCompile(`(\d{2})-(\d{2})`)
re.FindStringSubmatch("01-16")       // =>["01-16" "01" "16"]
re.FindStringSubmatch("01-16 04-21") // =>["01-16" "01" "16"]
re.FindStringSubmatch("01 16")       // =>[]
```

如果对应编号的捕获分组匹配的是空文本，则返回数组中对应元素为空字符串。

17.3.8　FindAllStringSubmatch

这个函数也可以视作上一个函数的完整版本，不同的是，它返回一个二维数组，其中的每个元素可以视为“当次匹配时调用 *FindStringSubmatch* 的结果”。它同样需要传入一个 int 参数，指定返回数组的大小：如果 $n>0$，则返回的数组中最多包含 n 个元素（更多匹配结果会被忽略）；如果 $n=0$，则返回空数组；如果 $n<0$，则返回所有的匹配。

```
re := regexp.MustCompile(`(\d{4})-(\d{2})-(\d{2})`)
re.FindAllStringSubmatch("2018-01-16 2017-12-27", 1)
// => [["2018-01-16" "2018" "01" "16"]]
re.FindAllStringSubmatch("2018-01-16 2017-12-27", 0)
// => []
re.FindAllStringSubmatch("2018-01-16 2017-12-27", -1)
// => [["2018-01-16" "2018" "01" "16"] ["2017-12-27" "2017" "12" "27"]]
```

17.3.9　SubexpNames

前面说过，Golang 的正则表达式支持命名分组。如果用到了命名分组，多半要用到 *SubexpNames* 这个函数，因为即便正则表达式中包含命名分组，匹配时还是需要调用 *FindStringSubmatch* 或者 *FindAllStringSubmatch*。然而此时的返回结果中并不包含命名分组的信息，所以需要手工“拼装”，将命名分组信息和 *FindStringSubmatch* 或者 *FindAllStringSubmatch* 的返回结果对应起来。

```
re := regexp.MustCompile(`(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})`)
names := re.SubexpNames()
// names => ["" "year" "month" "day"]，其实是各命名分组对应的数字编号
matches := re.FindStringSubmatch("2018-01-24")

md := map[string]string{}
for i, n := range matches {
    fmt.Printf("%d. match='%s'\tname='%s'\n", i, n, names[i])
    md[names[i]] = n
```

```
}

0. match='2018-01-24'  name=''
1. match='2018'        name='year'
2. match='01'          name='month'
3. match='24'          name='day'
```

17.3.10 Split

Split 可以用来切分字符串，它接收两个参数，第一个是希望切分的字符串——注意，是string，没有 byte[]的版本；第二个是 int 参数 *n*，它影响切分的结果：如果 *n*>0，返回的数组最多包含 *n* 个元素（最后一个元素对应的字符串不再做切分）；如果 *n*=0，则返回的是空数组；如果 *n*<0，则做尽可能多的切分。

```
re := regexp.MustCompile(`\d+`)
re.Split("A1B2C3D4", -1)     // =>["A" "B" "C" "D" ""]
re.Split("A1B2C3D4E", -1)    // =>["A" "B" "C" "D" "E"]
re.Split("A1B2C3D4E", 0)     // =>[]
re.Split("A1B2C3D4E", 2)     // =>["A" "B2C3D4E"]
re.split("A1B2C3D4E", 3)     // =>["A" "B" "C3D4E"]
```

要注意的是，Split 的结果**不会自动忽略空字符串**，所以如果你觉得结果有点怪异，很可能是空字符串的原因。

```
re := regexp.MustCompile(`\d+`)
re.split("A1B2C3D4", -1)     // =>["A" "B" "C" "D" ""]
```

17.3.11 ReplaceAllString

这个函数是进行替换操作的，它接收两个参数，第一个是需要进行替换处理的字符串，第二个是要替换成的字符串 *replacement*。

```
re := regexp.MustCompile(`/`)
re.ReplaceAllString("01/29/2018", "-")     // =>"01-29-2018"
```

也可以在 *replacement* 中用*$num* 的方式指定编号为 *num* 的捕获分组对应的文本。

```
re := regexp.MustCompile(`(\d{2})/(\d{2})/(\d{4})`)
re.ReplaceAllString("01/29/2018", "$3-$1-$2")     // =>"2018-01-29"
```

为了避免$的二义性，更好的做法是给*$num* 加上花括号，变成*${num}*。

```
re := regexp.MustCompile(`(\d{2})/(\d{2})/(\d{4})`)
re.ReplaceAllString("01/29/2018", "${3}-${1}-${2}")        // =>"2018-01-29"
//会产生二义性问题
re.ReplaceAllString("01/29/2018", "Y$3M$1D$2")             // =>"Y29"
```

```
//消除二义性
re.ReplaceAllString("01/29/2018", "Y${3}M${1}D${2}")       // =>"Y2018M01D29"
```

如果要在 *replacement* 中使用$字符，必须写作**$$**。

```
re := regexp.MustCompile(`\d+`)
//第一个$$代表$符号，之后的${0}表示表达式匹配的所有文本
re.ReplaceAllString("the price is 2199", "$$${0}")     // =>"the price is $2199"
```

17.3.12　ReplaceAllLiteralString

这个函数和上面的 *ReplaceAllString* 非常类似，是进行替换操作的，它接收两个参数，第一个是需要进行替换处理的字符串，第二个是要替换成的字符串 *replacement*。不同之处在于，*replacement* 中没有任何字符有特殊含义，全部被当成普通文字来处理。

```
re := regexp.MustCompile(`/`)
re.ReplaceAllLiteralString("01/29/2018", "-")      // =>"01-29-2018"
```

如果在 *replacement* 中使用*$num* 和**$$**等表示法，并不会产生特别的后果。

```
re := regexp.MustCompile(`(\d{2})/(\d{2})/(\d{4})`)
re.ReplaceAllLiteralString("01/29/2018", "$$-$3-$1-$2")     // =>"$$-$3-$1-$2"
```

17.4　常用操作示例

17.4.1　验证

简单验证

```
//注意要加上\A 和\z
regexp.MustCompile(`\A\d{4}-\d{2}-\d{2}\z`).MatchString("2018-01-31")  // => true
```

17.4.2　提取

```
re := regexp.MustCompile(`\d{4}-\d{2}-\d{2}`)
dates := re.FindAllString("2017-12-27 2018-01-31", -1)
for _, date := range dates {
    fmt.Println(date);
}
2017-12-27
2018-01-31

re := regexp.MustCompile(`(\d{4})-(\d{2})-(\d{2})`)
dates := re.FindAllStringSubmatch("2017-12-27 2018-01-31", -1)
for _, d := range dates {
```

```
    fmt.Printf("date:%s, year:%s, month:%s, day:%s\n", d[0], d[1], d[2], d[3]);
}
date:2017-12-27, year:2017, month:12, day:27
date:2018-01-31, year:2018, month:01, day:31
```

```
re := regexp.MustCompile(`(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})`)
names := re.SubexpNames()
//ni 表示 named capture index
ni := map[string]int{}
for index, name := range names {
    ni[name] = index
}
dates := re.FindAllStringSubmatch("2017-12-27 2018-01-31", -1)
for _, date := range dates {
    fmt.Printf("date:%s, year:%s, month:%s, day:%s\n",
        date[0], date[ni["year"]], date[ni["month"]], date[ni["day"]])
}
date:2017-12-27, year:2017, month:12, day:27
date:2018-01-31, year:2018, month:01, day:31
```

17.4.3 替换

简单替换

```
re := regexp.MustCompile(`(\d{2})/(\d{2})/(\d{4})`)
fmt.Println(re.ReplaceAllString("01/29/2018", "${3}-${1}-${2}"))
2018-01-29
```

在 replacement 中使用$

```
re := regexp.MustCompile(`\d+`)
fmt.Println(re.ReplaceAllString("the price is 2199", "$$${0}"))
the price is $2199
```

17.4.4 切分

简单切分

```
re := regexp.MustCompile(`\W+`)
fmt.Println(re.ReplaceAllString("one--two---three", -1))
[one two three]
```

第 18 章　Linux/UNIX

本章介绍 Linux 下常用工具（vi、grep、awk、sed）中正则表达式的应用。虽然大多数工具在 Linux/UNIX/FreeBSD 中都存在，但版本和用法存在若干差异。本章的讲解以 Linux 为准，如果你使用的是其他平台，可以参考本章的内容，具体细节需要参考对应的文档。

18.1　POSIX

如果你用过 vi、grep、awk、sed 之类的工具，或许会记得：这些工具虽然支持正则表达式，但语法却很不一样，按照通常习惯的方法写的正则表达式往往不是无法识别（`(abc)`竟然可能报错）就是匹配错误（`\d`无法匹配数字字符）。而且，这些工具自身之间也存在差异，同样的结构，有时需要转义有时不需要转义。这究竟是为什么呢？

原因在于，Linux/UNIX 下的工具大多采用 POSIX 规范，而 POSIX 规范又可分为两种流派（flavor）。之前只是简单介绍了 POSIX 字符组，现在详细讲解 POSIX 规范。

18.1.1　POSIX 规范

我们之前介绍的正则表达式的记法，其实都源于 Perl，实际上，正则表达式从 Perl 衍生出一个显赫的流派，叫作 PCRE（Perl Compatible Regular Expression），`\d`、`\w`、`\s`之类的字符组简记法是它的特征。但是在 PCRE 之外，正则表达式还有其他流派，比如 POSIX 规范的正则表达式。

POSIX 的全称是 Portable Operating System Interface for uniX，它由一系列规范构成，定义了 UNIX 操作系统应当支持的功能，所以“POSIX 规范的正则表达式”其实只是“关于正则表达式的 POSIX 规范”，它定义了 BRE（Basic Regular Expression，基本型正则表达式）和 ERE（Extended Regular Expression，扩展型正则表达式）两大流派。在兼容 POSIX 的 UNIX 系统上，grep 和 egrep 之类的工具都遵循 POSIX 规范，一些数据库系统中的正则表达式也符合 POSIX 规范。

18.1.1.1　BRE

grep、vi、sed 属于 BRE 这一派，它的语法看起来比较奇怪，元字符`(`、`)`、`{`、`}`必须转义之

后才具有特殊含义，比如(a)b 只能匹配 (a)b 而不是 ab；a{1,2}只能匹配字符串 a{1,2}，a\{1,2\}才能匹配字符串 a 或者 aa。

之所以这么麻烦，是因为正则表达式的许多功能是逐步出现的，一开始这些元字符并没有特殊的含义。为保证向后兼容，就只能使用转义。基于同样的原因，BRE 不支持+和?量词，也不支持多选结构(…|…)和反向引用\1、\2 等。

缺少了这些功能，BRE 使用起来确实不够方便，好在今天纯粹的 BRE 已经很少见了，毕竟大家已经认为正则表达式“理所应当”支持多选结构和反向引用等功能。所以虽然 vi 属于 BRE 流派，但仍然提供了这些功能。而且，GNU 也对 BRE 做了扩展，支持+、?、|，只是使用时必须写成\+、\?、\|，而且也支持\1、\2 之类的反向引用。这样，GNU 的 grep 等工具虽然名义上属于 BRE 流派，但更确切的名称是 GNU BRE。

18.1.1.2 ERE

egrep、awk 则属于 ERE 这一派，虽然从名字上来看，BRE 是“基本”而 ERE 是“扩展”，但 ERE 不要求兼容 BRE 的语法，而是自成一体。因此其中的元字符不用转义（在元字符之前添加反斜线会取消其特殊含义），所以(ab|cd)就可以匹配字符串 ab 或者 cd，量词+、?、{n,m}可以直接使用。ERE 并没有明确规定支持反向引用，但是不少工具都支持\1、\2 之类的反向引用。

GNU 出品的 egrep 等工具就属于 ERE 流派（更准确的名字是 GNU ERE），但因为 GNU 已经对 BRE 做了不少扩展，所谓的 GNU ERE 其实只是一个说法，它的功能 GNU BRE 都有了，GNU ERE 的主要区别在于元字符不需要转义。

表 18-1 简要说明了几种POSIX流派的区别[1]（其实，现在的BRE和ERE在功能上并没有什么区别，主要的差异是在元字符的转义上）。

表 18–1 几种 POSIX 流派的区别

流派	说明	工具
BRE	(、)、{、}都必须转义使用，不支持+、?、|	grep、sed、vi（但 vi 支持多选结构和反向引用）
GNU BRE	(、)、{、}、+、?、|都必须转义使用	GNU grep、GNU sed
ERE	元字符不必转义，+、?、(、)、{、}、|可以直接使用，对\1、\2 之类的支持不确定	egrep、awk
GNU ERE	元字符不必转义，+、?、(、)、{、}、|可以直接使用，支持\1、\2 之类	grep–E、GNU awk

[1] 关于 ERE 和 BRE 的详细规范，可以参考 http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html。

18.1.2　POSIX 字符组

POSIX 流派与 PCRE 流派的一大区别是字符组。

通常（也就是 PCRE 流派的做法）我们把[…]称为“字符组”；但在 POSIX 中，它们叫作“方括号表达式”（bracket expression），其作用与规则同常说的“字符组”完全一样：在方括号中列出可以匹配的所有字符，比如`[abc]`，也支持横线表示范围，比如`[a-z]`，如果在方括号里的开头写脱字符`^`，则表示排除型字符组，比如`[^0-9]`。但是，POSIX 中的“方括号表达式”都不支持使用**\u***xxxx* 的形式表示 Unicode 字符。也就是说，遵循 POSIX 规范的正则表达式无法用`[\u4E00-\u9FFF]`来匹配“任意一个中文字符”。

POSIX 自己也定义了字符组，准确的名称是“POSIX 字符组”（POSIX character class），这是一些事先定义好的组合，基本等价于我们常说的“字符组简记法”。比如`[:digit:]`就是一个 POSIX 字符组，它等价于`[0-9]`，`[:alpha:]`也是 POSIX 字符组，它等价于`[a-zA-Z]`。另一方面，PCRE 中常见的`\d`、`\w`、`\s`在 POSIX 中并没有定义。

POSIX 字符组不能直接使用，只能用于方括号表达式内部，类似`[[:digit:]]`和`[[:alpha:]]`；POSIX 字符组也可以与其他方括号表达式合用，比如`[a-fA-F[:digit:]]`；`[^[:alpha:]]`匹配一个非字母字符，`[^a-fA-F[:digit:]]`匹配一个十六进制数字之外的字符。

表 18-2 详细介绍了常用的 POSIX 字符组（请注意，POSIX 字符组只匹配 ASCII 字符）。

表 18–2　常用的 POSIX 字符组

POSIX 字符组	Perl 记法	ASCII 编码能匹配的字符	说明
`[:alnum:]`	类似`\w`，但不能匹配下画线	`[A-Za-z0-9]`	数字和字母字符
`[:alpha:]`		`[A-Za-z]`	字母字符
`[:ASCII:]`		`[\x00-\x7F]`	ASCII 字符
`[:blank:]`		`[ \t]`	空格和制表符
`[:cntrl:]`		`[\x00-\x1F\x7F]`	控制字符
`[:digit:]`	`\d`	`[0-9]`	数字字符
`[:graph:]`		`[\x21-\x7E]`	可见字符
`[:lower:]`		`[a-z]`	小写字母字符
`[:print:]`		`[\x20-\x7E]`	可见字符和空白字符
`[:punct:]`		`[][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]`	标点符号
`[:space:]`	`\s`	`[ \t\r\n\v\f]`	空白字符

（续表）

POSIX 字符组	Perl 记法	ASCII 编码能匹配字符	说明
`[:upper:]`		`[A-Z]`	大写字母字符
`[:xdigit:]`		`[A-Fa-f0-9]`	十六进制数字字符

了解了 POSIX、BRE、ERE 这些概念之后，下面简要介绍 Linux/UNIX 下常用工具中的正则表达式。

18.2 vi

vi/vim（vim 是 vi iMproved 的缩写，以下统称 vi）是 Linux/UNIX 下常用的文本编辑器，它提供了各种正则表达式功能。vi 的正则表达式属于 BRE 流派，但是它也支持几乎所有的正则表达式功能，只是不少元字符都需要转义。如果使用 Very Magic 模式（后面会讲到），也可以不用那么麻烦地进行转义。

首先介绍查找所用的各种记法。在 vi 里进行查找，首先要切换到控制模式（按 Esc 键），然后输入/*pattern*/（结尾的/可以忽略），其中 *pattern* 是用于查找的模式字符串。在控制模式下输入`/cat/`再回车，就查找当前编辑的文本中所有的字符串 cat（注意不是单词 cat）。

18.2.1 字符组及简记法

vi 的字符组遵循 POSIX 规范，所以 POSIX 中的各种字符组和记法都可以直接在 vi 中使用。如果在开方括号之前写上`\_`，则这个字符组不但能匹配其中列出的任何字符，还可以匹配行终止符，比如`\_[abc]`就不但能匹配字符 a、b、c，还能匹配换行符。

除去POSIX规范的字符组，vi也提供了类似Perl的字符组简记法。虽然vi在正确设定编码后[1]能够识别中文等多字节字符（不会把一个中文字符当成“几个字符”来处理），但其对“数字字符”“空白字符”和“单词字符”的识别仍然局限于ASCII字符，详见表 18-3。

表 18–3 vi 中的字符组简记法

字符组简记法	说明	字符组简记法	说明
`.`	除行终止符之外的任何字符	`\_.`	匹配任何字符（包括行终止符）
`\s`	ASCII 编码中的空白字符（但不包括行终止符）	`\S`	`\s` 无法匹配的字符

[1] 在 vi 中，切换到控制模式下，输入:set enc=xxx 可以设定编码，其中 xxx 可以是 utf8 或者 gbk，注意在 set 之前有一个冒号。

（续表）

字符组简记法	说明	字符组简记法	说明
\d	[0-9]	\D	\d 无法匹配的字符
\x	[0-9a-fA-F]	\X	\x 无法匹配的字符
\o	[0-7]	\O	\o 无法匹配的字符
\h	[a-zA-Z_]	\H	\h 无法匹配的字符
\p	可以打印的字符（非控制字符）	\P	除数字字符之外，所有的可打印字符
\w	[0-9a-zA-Z_]	\W	\w 无法匹配的字符
\a	[a-zA-Z]	\A	\a 无法匹配的字符
\l	[a-z]	\L	非小写字母字符
\u	[A-Z]	\U	非大写字母字符

补充一点，表中列出的字符组简记法是不能在方括号内部使用的。比如要匹配一个十六进制的数字字符，可以用\x，也可以用[[:digit:]a-fA-F]，但[\da-fA-F]无法匹配[0-9]。

值得注意的是，vi 中没有单行模式，所以点号不能匹配行终止符，如果需要匹配包括行终止符在内的任何字符，可以使用_.。

18.2.2　量词

vi 中的量词比较奇怪，大多需要转义，部分通行量词的记法也有区别（主要是{出现之后，}不需要转义），刚开始接触可能不习惯。表 18-4 列出了 vi 中的量词。

表 18–4　vi 中的量词

量词	等价的 PCRE 量词
*	*
\+	+
\=	?
\{n,m}	{n,m}
\{n}	{n}
\{n,}	{n,}
\{, m}	{0,m}
\{-}	*?
\{-n,m}	{n,m}?
\{-n,}	{n,}?
\{-,m}	{,m}?

vi 中没有现成的忽略优先量词`+?`和`??`，但是可以用`\{-1, }`和`\{-, 1}`表示。

18.2.3 多选结构和捕获分组

前面讲过，vi 的正则表达式属于 BRE 流派，所以`(`、`|`、`)`都需要转义。通常我们所见的`(…|…)`，要写作`\(…\|…\)`。

捕获分组中的括号也需要转义，所以`(…)`要写作`\(…\)`。

18.2.4 环视

vi 也提供了环视功能，但是它的环视写法有所不同。vi 的环视与 PCRE 的环视对照如表 18-5 所示。

表 18–5 vi 的环视与 PCRE 的环视对照举例

vi	PCRE
`\@=`	`?=`
`\@!`	`?!`
`\@<=`	`?<=`
`\@<!`	`?<!`

同时环视的写法也与常见的不同，PCRE 将环视使用的表达式写在环视的后方（右方），而 vi 中必须将环视使用的表达式写在环视的左方，表 18-6 举了两个例子。

表 18–6 vi 的环视与 PCRE 的环视对照举例

vi	PCRE
`ab\(cd\)\@=`	`ab(?=cd)`
`\(foo\)\@<!bar`	`(?<!foo)bar`

vi 的环视功能没有任何限制，可以随意使用量词，所以可以很方便地用它查找“包含/不包含某个表达式”的行。

查找包含 tom 或者 jerry 的行。

```
^\(.*(tom|jerry)\)\@=
```

查找不包含 tom 也不包含 jerry 的行。

```
^\(.*(tom|jerry)\)\@!
```

18.2.5 锚点和单词边界

vi 提供了常见的匹配行起始位置和结束位置的`^`和`$`，但没有提供“通用”的单词边界`\b`，而是提供了分别匹配单词起始和结束位置的`\<`和`\>`。详见表 18-7。

表 18–7　vi 中的锚点和单词边界

分界符	说明
^	匹配行的起始位置
$	匹配行的结束位置
\<	单词开始位置，左边不能是[0-9a-zA-Z_]，右边必须是[0-9a-zA-Z_]
\>	单词结束位置，右边不能是[0-9a-zA-Z_]，左边必须是[0-9a-zA-Z_]

下面举几个例子。

匹配行开始的 cat。

```
^cat
```

匹配行结束的 cat。

```
cat$
```

匹配单词开始的 cat。

```
\<cat
```

匹配单词结束的 cat。

```
cat\>
```

回到开头提到的查找 cat 的例子，如果我们要查找的是“单词”cat，就应该写作`\<cat\>`。

18.2.6　替换操作的特殊字符

vi 中的替换与查找类似，先切换到控制模式（按 Esc 键），然后输入`:`*x*`,`*y*`s/pattern/replacement/`。其中 *x* 和 *y* 用来限制替换操作所在的行数（从第 *x* 行开始，到第 *y* 行结束，如果不指定 *x* 和 *y*，则只在当前行进行替换；如果只写一个`%`，则对所有行进行替换），**s** 表示“替换”操作，*pattern* 是用来查找的模式字符串，*replacement* 则是要替换成的字符串，当然也不要忘记最前面的冒号。

在当前行将单词 tom 替换为 jerry。

```
:s/\<tom\>/jerry/
```

在第 1 行到第 10 行将单词 tom 替换为 jerry。

```
:1,10s/\<tom\>/jerry/
```

在所有行将单词 tom 替换为 jerry。

```
:%s/\<tom\>/jerry/
```

为简明起见，下面的替换命令中不再写出:x,y，只保留 s/pattern/replacement/，请读者注意。

在默认情况下，替换只会对第一次遇到的匹配进行，如果在某行中多次出现了正则表达式能够匹配的文本，需要将它们全部替换，可以使用 s/pattern/replacement/g，最后的 g 表示“全部替换”模式。

在所有行中将出现的所有单词 tom 替换为 jerry。

```
:%s/\<tom\>/jerry/g
```

在结尾的/之后，除了 g，还有其他模式，如表 18-8 所示。

表 18–8 vi 中能使用的模式

模式	说明
g	全局性替换（尽可能多地替换）
i	查找时不区分大小写
c	在替换前需要提示确认

下面再给出一些常见操作的写法。

在所有行中将所有出现的单词 tom（不区分大小写）替换为 jerry，并在替换前确认。

```
:%s/\<tom\>/jerry/igc
```

删除所有行开头的空白。

```
s/^\s\+//
```

删除所有行结尾的空白。

```
s/\s\+$//
```

删除所有只包含空白字符的行中的空白字符。

```
%s/^\s\+$//g
```

行开头添加#（Python 和 Ruby 注释）。

```
%s/^/#/
```

删除行末的注释#。

```
s/#\s\+$//
```

将连续空行合并为一个空行。

```
s/^\_[[:space:]]\+$//g
```

删除 HTML tag。

```
s/<[^>]\+>//g
```

删除所有包含 tom 或 jerry 的行。

```
s/^\(.*(tom|jerry)\)\@=.*$//i
```

删除所有//注释。

```
s/\/\/.*$//
```

18.2.7　replacement 中的特殊变量

表 18-9 中列出了 replacement 中可以使用的特殊变量。

表 18–9　replacement 中可以使用的特殊变量

变量	说明
&	表达式匹配的全部文本
\0	表达式匹配的全部文本
\1	编号为 1 的捕获分组匹配的文本
\2	编号为 2 的捕获分组匹配的文本
\L	将之后的所有字符转换为小写
\U	将之后的所有字符转换为大写
\E	\U 和\L 的作用结束
\e	\U 和\L 的作用结束
\l	将此后一个字符转换为小写
\u	将此后一个字符转换为大写

用<b>标签标注 word。

```
s/\<word\>/<b>&</b>/g
```

删除重复单词。

```
s/\<(\w+)\>\s\+\<\1\>/\1/g
```

找出所有形式的 tom 或者 jerry，全部转换为小写。

```
s/\<\(tom\|jerry\)\>/\L&/gi
```

将所有的 tom 或 jerry 统一为首字母大写的形式。

```
s/\<\(tom\|jerry\)\>/\u&/g
```

18.2.8　补充

习惯 PCRE 正则表达式的用户可能不习惯 vi 中频繁出现的转义。如果是这样，可以使用 vi

提供的 Very Magic 模式，在这种模式下，常用的$、.、*、(、)、|都不用转义，只有{和}需要转义。

Very Magic 模式的使用非常简单，在查找字符串的开始写上\v 就可以。表 18-10 中给出了两个例子。

表 18–10 Very Magic 模式使用举例

普通模式	Very Magic 模式
\(…|…\)	\v(…|…)
\(…\)	\v(…)

在 vi 中输入:h pattern 可以查询到 pattern 的详细信息，也可以参考这两份文档。

http://en.wikipedia.org/wiki/Vim_(text_editor)
http://vimdoc.sourceforge.net/

18.3 grep

grep 是 Linux/UNIX 下常用的工具，作用是从文件或标准输入中筛选出包含某个正则表达式能匹配文本的行，将其传送至标准输出。grep 这个名字来自它所模拟的 ed 编辑器的一系列操作的名称缩写：Global Regular Expression Print，也有一种说法是 Generalized Regular Expression Parser，最早出现在 1973 年的 UNIX Version 4 中。目前大多数 Linux/UNIX 中都提供了 grep（Windows 下也有 grep，但是需要自己下载），其中以 GNU grep 最为流行，下面的讲解就以 GNU grep 为准。

18.3.1 基本用法

grep 的基本用法是：

```
grep [options] PATTERN [FILES]
```

其中的 *options* 在本节最后介绍，*PATTERN* 是需要指定的正则表达式，*FILES* 是需要检索的文件，如果不指定 *FILES*，则默认从标准输入（stdin）读取文本，下面举一个最简单的例子。

从 sample.txt 中找出所有包含 word 的行。

```
grep word sample.txt
```

为了避免正则表达式被 Shell 错误解释（正则表达式中的*、[、]之类可能先被 Shell 展开），保险的做法（也是好的习惯）是给正则表达式加上单引号'。

```
grep 'word' sample.txt
```

如果在其中出现了变量插值，则应当使用双引号"。

```
grep "$JAVA_HOME" sample.txt
```

18.3.2　字符组

grep 的字符组遵循 POSIX 规范，所以 POSIX 的字符组和简记法都可以直接在 grep 中使用。grep 中的点号可以匹配除换行符之外的任何字符，因为 grep 是按行处理的，每次处理一行，所以不需要关心点号能否匹配换行符。

从 sample.txt 中找出所有这样的行：包含字符 a 或 b。

```
grep '[ab]' sample.txt
```

从 sample.txt 中找出所有这样的行：包含除 a 和 b 之外的任何字符。

```
grep '[^ab]' sample.txt
```

从 sample.txt 中找出所有这样的行：包含 c 开头，t 结尾，长度为 3（如 cat、cut 等）的字符串的文本行。

```
grep 'c.t' sample.txt
```

18.3.3　锚点和单词边界

在 grep 中能使用的锚点有 4 个：`^`、`$`、`\<`、`\>`。`^`用来匹配行的开始位置，`$`用来匹配行的结束位置（因为 grep 是面向行的处理工具，每次处理一行，所以不存在`^`和`$`匹配文本内部的行起始/结束位置的问题，也就不需要多行模式）。grep 不支持常见的`\b`，而是用`\<`和`\>`来匹配单词的起始位置和结束位置，它们所识别的“单词字符”是`[[:alnum:]]`能匹配的字符，也就是`[0-9a-zA-Z]`，注意其中不包含下画线。

从 sample.txt 中找出所有以 cat 开头的行。

```
grep '^cat' sample.txt
```

从 sample.txt 中找出所有以 cat 结尾的行。

```
grep 'cat$' sample.txt
```

从 sample.txt 中找出所有包含单词 cat 的行。

```
grep '\<cat\>' sample.txt
```

18.3.4　量词

grep 中能使用的量词有`*`、`+`、`?`、`{n, m}`，其中只有`*`可以直接使用，其他量词都必须转义：`+`必须写成`\+`，`?`必须写成`\?`，`{n, m}`要写成`\{n,m\}`。

从 sample.txt 中找出所有包含 a、ab、abb、abbb…的行。

```
grep 'ab*' sample.txt
```

从 sample.txt 中找出所有包含 ab、abb、abbb…的行。

```
grep 'ab\+' sample.txt
```

从 sample.txt 中找出所有包含 a 或者 ab 的行。

```
grep 'ab?' sample.txt
```

从 sample.txt 中找出所有包含 ab、abb、abbb 的行。

```
grep 'ab\{1,3\}' sample.txt
```

18.3.5 多选结构和捕获分组

grep 支持多选结构，但是必须转义`(`、`|`、`)`三个字符，多选结构必须写成`\(…\|…\)`，捕获分组的记法是`\1`、`\2`等。

查找 sample.txt 中所有包含 tom 或者 jerry 的行。

```
grep '\(tom\|jerry\)' sample.txt
```

查找 sample.txt 中所有包含重复单词的行。

```
grep '\(\<[[:alpha:]]\+\>\)[[:space:]]\+\<\1\>' sample.txt
```

18.3.6 options

grep 中可以使用 *options* 来指定匹配规则（类似匹配模式），常用的 *option* 介绍如下。

–i 不区分大小写

查找各种形式的 cat，包括 CAT、cat、Cat 等。

```
grep -i 'cat' sample.txt
```

–v 不匹配

这是 grep 中特别有用的功能，正则表达式要“排除若干字符串”通常是非常麻烦的，但是有了 `grep -v` 就非常方便了，它会选出正则表达式不能匹配的所有行。

查找所有不包含 cat 的行。

```
grep -i 'cat' sample.txt
```

查找所有不包含 404 和 302 的行（在过滤日志时特别有用）。

```
grep -v '\(404\|302\)' sample.txt
```

查找所有不包含 jpg、png、gif 的行（在过滤日志时特别有用）。

```
grep -v '\(jpg\|png\|gif\)' sample.txt
```

–w 只匹配单词

查找只包含单词 cat 的行，等价于 `grep '\<cat\>' sample.txt`。

```
grep -w 'cat' sample.txt
```

–n 显示行数

找出所有包含 cat 的行，同时标明行数。

```
grep -n 'cat' sample.txt
```

–r 递归查找

这也是一个特别有用的功能，它可以对目录下的所有文件进行查找。

找出当前目录下所有文件中包含 cat 的行。

```
grep -r 'cat' .
```

–A *n* 显示匹配行及之后的 *n* 行

找出所有包含 cat 的行，并显示每行之后的 3 行。

```
grep -A 3 'cat'
```

–B *n* 显示匹配行及之前的 *n* 行

找出所有包含 cat 的行，并显示每行之前的 3 行。

```
grep -B 3 'cat'
```

–C *n* 显示匹配行及之前之后各 *n* 行（等价于–A *n* –B *n*）

找出所有包含 cat 的行，并显示每行之前和之后的 3 行。

```
grep -C 3 'cat'
```

–P 使用 Perl 风格的表达式

如果你很熟悉 Perl 风格的正则表达式，而不习惯 grep 的规定，开启这个选项可以直接使用 Perl 风格的正则表达式，比如像下面这样查找单词 cat。

```
grep -P '\bcat\b'
```

18.3.7　egrep 和 fgrep

egrep 和 grep 的名字不同，但并没有本质区别。虽然 egrep 属于 ERE 流派，而 grep 属于 BRE 流派，但 GNU egrep 和 GNU grep 其实只有写法的差别，功能上并无不同。实际上，在命令行下输入 grep –E，就等价于 egrep。使用 egrep 的理由或许在于，因为它属于 GNU ERE 流派，元字符几乎不需要转义就可直接使用，写、读起来都要更简单一些。

fgrep 也是 grep 家族中的成员，f 代表 fast，fgrep 也确实比 grep 要快很多，因为它完全取消了正则表达式的功能，只进行最简单的字符串查找。在 fgrep 中查找 `end$`，并不是找出行末为

end 的行，而是包含字符串 `end$`的行。

18.3.8 补充

grep 负责选出能匹配成功的整行文本，真正匹配到的内容反而不好识别了。使用 grep 时最好加上一个参数 `grep --color`，这样 grep 会把能匹配到的文本用不同颜色标注出来，方便识别。如果你愿意，不妨做一个 alias，省去每次明确指定：

```
alias grep='grep --color'
```

关于 grep 的详细文档，可以输入 `info grep` 获得，也可以参考下面的文档：

http://www.gnu.org/software/grep/manual/
http://en.wikipedia.org/wiki/Grep

18.4 awk

awk 也是 Linux/UNIX 下常用的工具，它在 20 世纪 70 年代诞生于贝尔实验室，得名自三位作者 Alfred Aho、Peter Weinberger、Brian Kernighan。awk 的主要作用是对数据重新组织和统计，它的功能非常多，语法也很复杂，如果使用得当完全可以实现非常复杂的逻辑。在本节，我们只介绍与正则表达式相关的内容。awk 有许多版本，本章以 gawk（GNU awk）为准。

18.4.1 基本用法

awk 的基本使用方法是：

```
awk 'program' input-file1 input-file2 ...
```

其中 *program* 是处理数据的程序代码，其基本形式为：

condition { *action* }

condition 是判断条件，*action* 是条件满足时所执行的操作，比如下面这条命令的意思就是，如果 sample.txt 中的某一行可以由正则表达式 `J` 匹配（也就是说，包含大写字母 J），就输出这一整行（`$0` 的含义在下面讲解）。

```
awk '/J/ {print $0}' sample.txt
```

为方便讲解，假设 sample.txt 的内容如下：

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
```

```
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
```

上面命令运行的结果就是：

```
Jan  13  25  15 115
Jun  31  42  75 492
Jul  24  34  67 436
```

如果仅仅这样使用，awk 和 grep 并没有太大的区别，但 awk 的功能并不限于这些。awk 提供了一系列变量$1、$2、$3…对应每一行的各个字段（在默认状态下，字段之间以空白分隔，也可以通过 FS 指定分隔），$0 代表整行，NF 代表该行的字段数目，因此$NF 表示最后一个字段。

如果把上面的语句改一改，就可以同时实现对字段的过滤。

```
awk '/J/ {print $1, $2, $NF}' sample.txt

Jan  13 115
Jun  31 492
Jul  24 436
```

我们还可以在{*action*}中写上一些处理逻辑。

```
awk '{if( $NF > 400) print $0}' sample.txt

Apr  31  52  63 420
Jun  31  42  75 492
Jul  24  34  67 436
```

下面介绍 awk 中的正则表达式。awk 中的正则表达式虽然也遵循 POSIX 规范，但属于 ERE 流派，所以很多元字符不需要进行转义。

18.4.2　字符组及简记法

awk 的字符组遵循 POSIX 规范，所以 POSIX 的字符组和简记法都可以直接在 awk 中使用。而且，awk 中的点号可以匹配包括换行符在内的任何字符，不过一般来说，awk 是按行处理的，所以这一点并不重要。

从 sample.txt 中找出所有包含 a 或者 b 的行。

```
awk '/[ab]/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含数字字符的行。

```
awk '/[0-9]/ {print $0}' sample.txt
awk '/[[:digit:]]/ {print $0}' sample.txt
```

例外的是，GNU awk 支持两个 PCRE 的字符组\w和\W，其中\w等价于[[:alnum:]_]，\W则匹配\w所不能匹配的所有字符。

18.4.3 锚点和单词边界

awk 中能使用的锚点有下面几个：^、$、\<、\>、\y、\B。^用来匹配行的开始位置，$用来匹配行的结束位置（因为 awk 是面向行的处理工具，每次处理一行，所以不存在^和$匹配文本内部的行起始和结束位置的问题，也就不需要多行模式）。

awk 识别的“单词字符”是[[:alnum:]]能匹配的字符，也就是[0-9a-zA-Z]，用\<和\>匹配单词的起始和结束位置，\y匹配单词的起始或者结束位置，\B匹配\y无法匹配的位置。

从 sample.txt 中找出所有以 cat 开头的行。

```
awk '/^cat/ {print $0}' sample.txt
```

从 sample.txt 中找出所有以 cat 结尾的行。

```
awk '/cat$/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含以 cat 开头单词的行。

```
awk '/\<cat/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含以 cat 结尾单词的行。

```
awk '/cat\>/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含单词 cat 的行。

```
awk '/ \<cat\>/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含 cat，但不是单词 cat 的行（比如 truncate）。

```
awk '/\Bcat\B/ {print $0}' sample.txt
```

18.4.4 量词

awk 中能使用的量词有*、+、?、{n,m}、{n,}、{n}，因为 awk 属于 ERE 流派，所以这些量词可以直接使用，不用转义。

从 sample.txt 中找出所有包含 a、ab、abb、abbb…的行。

```
awk '/ ab*/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含 ab、abb、abbb…的行。

```
awk '/ ab+/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含 a 或者 ab 的行。

```
awk '/ ab?/ {print $0}' sample.txt
```

从 sample.txt 中找出所有包含 ab、abb、abbb 的行。

```
awk '/ ab{1,3}/ {print $0}' sample.txt
```

18.4.5　多选结构

awk 中的多选结构也是`(…|…)`的形式，可以直接使用，不需要转义。

从 sample.txt 中找出包含 tom 或者 jerry 的行。

```
awk '/(tom|jerry)/ {print $0}' sample.txt
```

18.4.6　补充

awk 不支持捕获分组和反向引用，不支持忽略优先量词，也无法使用匹配模式进行不区分大小写的查找。

关于 awk 的详细文档，可以输入 `info awk` 获得，也可以参考下面的文档：

http://www.gnu.org/manual/gawk/gawk.html

18.5　sed

sed 也诞生于贝尔实验室，是 Linux/UNIX 下常用的流编辑工具。sed 得名自 stream editor，它主要用来进行文本的替换等操作。因为 sed 进行流式处理（在默认模式下，sed 每次读入一行，去掉换行符，处理，再加上换行符，输出），所以速度非常快。sed 有许多版本，这里讲解的以 GNU sed 为准（选择它的理由是因为它比较标准，也很方便，GNU sed 提供了不区分大小写的 i 模式，其他的一些 sed 则没有提供）。前面讲过，sed 属于 BRE 流派，所以许多元字符之前需要加反斜线来转义。

18.5.1　基本用法

sed 最主要的功能是进行文本的替换，替换的基本用法如下：

```
sed –e 's/pattern/replacement/options' filename
```

其中 `s` 表示“替换”，*pattern* 是要用来搜索替换文本的正则表达式，而 *replacement* 是要替换的结果，*filename* 是要操作的文件，*options* 是可选参数。如果必须同时执行多个替换操作，则每个操作之前都必须使用`-e`。

```
sed –e 's/pattern1/replacement1/' –e 's/pattern2/replacment2/' filename
```

在默认情况下，sed 只会对每行文本进行一次替换，如果要替换所有的文本，则应该设定 *options* 参数为 g：

```
sed -e 's/pattern/replacement/g' filename
```

18.5.2 字符组及简记法

sed 的字符组遵循 POSIX 规范，所以 POSIX 的字符组和简记法都可以直接在 sed 中使用。需要指出的是，sed 中的点号.可以匹配包括换行符在内的任何字符，不过一般来说，sed 是按行处理的，所以这点区别并不重要。

从 sample.txt 中删除所有 a 或者 b。

```
sed -e 's/[ab]//' sample.txt
```

从 sample.txt 中删除所有数字字符。

```
sed -e 's/[[:digit:]]//g' sample.txt
sed -e 's/[0-9]//g' sample.txt
```

18.5.3 锚点和单词边界

sed 支持^和$，其中^匹配行的起始位置，$匹配行的结束位置。

从 sample.txt 中替换掉所有在行首的 cat

```
sed -e 's/^cat//g' sample.txt
```

从 sample.txt 中替换掉所有在行尾的 cat

```
sed -e 's/cat$//g' sample.txt
```

sed 认定的“单词字符”是\w，也就是[[:alnum:]_]，在此基础上，共有 4 个单词边界：\<、\>、\b、\B。\<匹配单词的起始位置，\>匹配单词的结束位置，\b 匹配单词的起始或者结束位置，\B 则匹配\b 无法匹配的位置。

从 sample.txt 中替换所有单词开头的 cat。

```
sed -e 's/\<cat//g' sample.txt
```

从 sample.txt 中替换所有单词末尾的 cat。

```
sed -e 's/cat\>//g' sample.txt
```

从 sample.txt 中替换所有的单词 cat。

```
sed -e 's/\bcat\b//g' sample.txt
sed -e 's/\<cat\>//g' sample.txt
```

从 sample.txt 中替换所有单词内部的 cat（比如 truncate）。

```
sed -e 's/\Bcat\B//g' sample.txt
```

18.5.4　量词

前面讲过，sed 属于 BRE 流派，所以虽然能使用的量词有*、+、?、{n, m}、{n}、{n,}，但只有*可以直接使用，其他量词都必须转义：+必须写成\+，?必须写成\?，{和}都需要转义。

从 sample.txt 中删除所有的 a、ab、abb、abbb 等。

```
sed -e 's/ab*//g' sample.txt
```

从 sample.txt 中删除所有的 ab、abb、abbb 等。

```
sed -e 's/ab\+//g' sample.txt
```

从 sample.txt 中删除所有的 a 或者 ab。

```
sed -e 's/ab\?//g' sample.txt
```

从 sample.txt 中删除所有的 ab、abb、abbb。

```
sed -e 's/ab\{1,3\}//g' sample.txt
```

18.5.5　多选结构和捕获分组

sed 中多选结构的(、|和)都需要转义，写成\(…\|…|)。

从 sample.txt 中替换掉所有的 tom 或者 jerry。

```
sed -e 's/\(tom\|jerry\)//g' sample.txt
```

如果出现了捕获分组，则无论在 regex 中，还是 replacement 中，都可以用\1、\2 的形式来引用。

从 sample.txt 中删掉重复的单词。

```
sed -e 's/\(\b[[:alnum:]]\+\b\)[[:space:]]\+\1//g' sample.txt
```

从 sample.txt 中把重复的单词替换为单个单词。

```
sed -e 's/\(\b[[:alnum:]]\+\b\)[[:space:]]\+\1/\1/g' sample.txt
```

sed 中有一个特殊变量&，它表示正则表达式匹配的文本。这样，我们可以给所有的数字字符串加粗：

```
sed -e 's/[[:digit:]]\+/<b>&<\/b>/g' sample.txt
```

18.5.6 options

sed 的主要 *option* 介绍如下。

g：在所有可能的地方进行替换

把 sample.txt 中所有的 tom 替换为 jerry。

```
sed -e 's/tom/jerry/g'
```

p：输出发生了替换的行。

它配合-n 使用，可以只输出发生了替换的行。

把 sample.txt 中所有的 tom 替换为 jerry，并且只输出发生了替换的行。

```
sed -n -e 's/tom/jerry/gp'
```

I：查找-替换时不区分大小写（只有 GNU sed 支持此功能）

把 sample.txt 中所有的 tom（不区分大小写）替换为 jerry。

```
sed -e 's/tom/jerry/Ig' sample.txt
```

M：启用多行模式（只有 GNU sed 支持此功能）

在默认情况下，sed 按行读入文本，然后处理，但我们也可以指定 N 参数，让 sed 一次读入多行文本，此时如果指定了 M 模式，^和$就能匹配多行文本中某一行的起始和结束位置。

一次读入两行，将行首的 tom 替换为 jerry（不要忘记 g，因为要处理“所有的行首”）。

```
echo -e 'tom\ntom' | sed 'N; s/^tom/jerry/Mg
jerry
jerry
```

18.5.7 补充

如果觉得 BRE 的转义太麻烦，也可以指定使用 ERE 模式，办法就是在 sed 之后添加-r 参数，如果之前使用了-e 参数，可以把两个参数合并为-re（写成-er 会报错）。

```
BRE
sed -e 's/\(\b[[:alnum:]]\+\b\)[[:space:]]\+\1\b/\1/' sample.txt
ERE
sed -re 's/(\b[[:alnum:]]+\b)[[:space:]]+\1\b/\1/' sample.txt
```

sed 不支持环视，也不支持忽略优先量词。

关于 sed 的详细文档，可以输入 `info sed` 获得，也可以参考下面的文档：

http://www.gnu.org/software/sed/manual/sed.html

http://www.linuxtopia.org/online_books/linux_tool_guides/the_sed_faq/index.html

18.6　总结

本章介绍了 Linux/UNIX 下几款常用的文本编辑工具，它们的功能各不相同，但又有所重叠，为方便大家理解，表 18-11 列出了各款工具适合解决的问题类型。

表 18–11　几种工具的总结

工具	特性	适用问题
vi	全文编辑工具，通用文本编辑	全文编辑，微调整篇文本
grep	流编辑工具，不能修改文本，只能进行查找，结果中仍然保留原来行的形态	在大量文本中找出感兴趣的行，日志的筛选
awk	流编辑工具，不能修改文本，但结果不必保留原来行的形态，可以截取、重新组织各个字段，编程能力强大，可设定变量	筛选出感兴趣的字段，转换数据格式，生成统计报告
sed	流编辑工具，可以大幅度修改文本	批量修改大量文本（比如修改源代码中的链接），做临时修改配合 awk 和 grep 的使用

因为在实际开发中我们可能面对海量数据，用任何一款全文编辑器（包括vi）打开编辑都是不现实的，只能用流编辑器来处理，而且必须混用grep、awk、sed，让它们各司其职，配合完成任务。常见的流程是，先用grep筛选出真正感兴趣的行，再用sed修改数据、awk整理格式并进行统计。[1]

下面给出一个真实的例子，作为本章的结尾。

某服务器记录了客户端软件的访问记录，其日志样本如下：

```
    server1 - - [20/Dec/2010:06:33:58 +0800] "POST /get HTTP/1.1" 200 417 "-" "program-win/
0.6.0.253 (b:i386; o:Microsoft Windows 7 Ultimate Edition (build 7600), 64-bit)"
"219.70.179.147"
    server1 - - [20/Dec/2010:06:33:59 +0800] "POST /get HTTP/1.1" 200 839 "-" "program-win/
0.6.0.240 (b:i386; o:Microsoft Windows 7 Ultimate Edition Service Pack 1, v.721 (build
7601), 64-bit)" "122.70.32.72"
    server1 - - [20/Dec/2010:06:33:59 +0800] "POST /get HTTP/1.1" 200 839 "-" "program-win/
0.6.0.240 (b:i386; o:Microsoft Windows 7 Ultimate Edition Service Pack 1, v.721 (build
7601), 64-bit)" "122.70.32.72"
    server1 - - [20/Dec/2010:07:13:27 +0800] "POST /get HTTP/1.1" 200 1078 "-" "program-
linux/0.6.0.253 (X11; U; Linux x86_64; en-US; rv:1.9.2.13) " "125.114.169.249"
```

[1] awk 也可以执行 grep 的“筛选”功能，但 grep 可以智能混用正则表达式匹配和字符串的 Boyer-Moore 搜索算法，在进行简单处理时不需要动用正则表达式匹配，所以速度远远高于其他“纯”正则表达式工具。因此，进行文本筛选时，grep 是首选工具。

每一行记录一次访问，包含的信息有：所访问的服务器、日期、HTTP Header、时间、版本、客户平台、客户端 IP。

要完成的任务是统计 Windows 程序的版本分布状况，生成只包含版本号（不包含平台信息）和客户端 IP 地址的数据文件。上面的日志中有 3 行记录了 Windows 程序的访问，最后得到的数据文件如下：

```
0.6.0.253 219.70.179.147
0.6.0.240 122.70.32.72
0.6.0.240 122.70.32.72
```

一般来说，此类任务应当首先用 grep“筛选”出真正要处理的行（也就是包含 program-win 的行），然后用 sed 处理。常见的直接思路是用两个捕获分组匹配要提取的字段，将整行替换为这两个捕获分组的内容。整个程序如下：

```
grep 'program-win' log \|
sed -re 's/^.*'program-win\/([^ ]+).*"([^"]+)"$/\1 \2/'
```

注意：这里用了-r 选项，使用 ERE 写法，否则要增加不少元字符转义。

```
grep 'program-win' log \|
sed -re 's/^.*'program-win\/\([^ ]\+\).*"\([^"]\+\)"$/\1 \2/'
```

思路是使用捕获分组抓取所需要的数据，将整行替换为这两个分组捕获的内容。结果没有错，但速度太慢，处理 200 万条数据需要 2 分钟左右，在生产环境中不可接受。

如果仔细思考就会发现，真正的问题只是“将感兴趣的内容提取出来，忽略其他内容”，捕获分组和替换并非必需。如果能直接把所需要的数据提取出来，就可以节省很多时间。

但是，awk 的切分必须使用统一的分隔符，但要提取所需要的数据，有时需要以空白字符作为字段分隔，有时必须以/字符作为字段分隔，所以仅仅依靠 awk 是不行的。这时候，不妨动用 sed，将反斜线字符替换为空格，再切分。以 grep 提取出的第一行为例，原始数据是这样的：

```
server1 - - [20/Dec/2010:06:33:58 +0800] "POST /get HTTP/1.1" 200 417 "-" "program-win/0.6.0.253 (b:i386; o:Microsoft Windows 7 Ultimate Edition (build 7600), 64-bit)" "219.70.179.147"
```

先用双引号"作为分隔符，切分出完整的版本信息和客户端 IP 地址。

```
awk 'BEGIN { FS = "\"" } ; { print $6, $8 }'
```

```
program-win/0.6.0.253 (b:i386; o:Microsoft Windows 7 Ultimate Edition (build 7600), 64-bit) 219.70.179.147
```

这时候把反斜线替换为/空格，再切分，提取出第 2 个字段，以及最后一个字段（使用特殊变量$NF）就可以完成：

```
sed -e 's/\// /g' | awk '{print $2, $NF}'
0.6.0.253 219.70.179.147
```

最后把这些操作整合起来：

```
grep 'program-win' log \|
awk 'BEGIN { FS = "\"" } ; { print $6, $8 }' \|
sed -e 's/\// /g' | awk '{print $2, $NF}'
```

这样避免了使用捕获分组，而且大大减少了替换操作的工作量。经过测试，200 万条数据只用不到 10 秒就可以处理完成，速度提高了 10 倍以上。

从这个例子中我们可以看到 grep、awk、sed 适合解决的问题：grep 适合过滤和筛选，awk 适合重新组织数据，sed 可以修改数据，也可以整理格式，为 awk 的处理提供便利。妥善结合运用这三种工具，可以大大提高解决问题的效率。

附录 A　常用语言中正则特性一览

字符及字符组								
特性	.NET	Java	JavaScript	PHP	Python	Ruby	Objective–C	Golang
\Q…\E	×	√	×	√	×	×	√	√
\d\w\s 字符组简记法	①	ASCII 匹配规则	ASCII 匹配规则，\s 除外	ASCII 匹配规则	②	ASCII 匹配规则	Unicode 匹配规则	ASCII 匹配规则
POSIX 字符组	×	ASCII 字符	×	ASCII 字符	×	③	×	ASCII 字符

①可以指定RegexOptions.ECMAScript恢复到ASCII匹配规则

②Python 2 默认采用ASCII匹配规则，但指定Re.U模式可以变换为Unicode匹配规则；Python 3 默认采用Unicode匹配规则，但指定Re.A模式可以变换为ASCII匹配规则

③Ruby 1.9 的POSIX字符组支持Unicode字符，且不需要显式指定Unicode模式

括号相关								
特性	.NET	Java	JavaScript	PHP	Python	Ruby	Objective–C	Golang
表达式中\1\9 引用分组	√	√	√	√	√	√	√	×
替换中引用分组	$*num*	$*num*	$*num*	$*num*	\g<*num*>	*num*	$*num*	$*num*
命名分组	①	×	×	②	③	④	⑤	⑥

①.NET中用(?<*name*>regex)表示命名分组，用\k<*name*>在表达式中引用分组，用${*name*}在替换中引用

②PHP中用(?P<*name*>regex)表示命名分组，用(P=*name*)在表达式中引用分组，替换中不能引用命名分组

③Python中用(?P<*name*>regex)表示命名分组，用(P=*name*)在表达式中引用分组，用\g<*name*>在替换中引用

④只有Ruby 1.9 支持命名分组，用(?<*name*>regex)表示命名分组，用\k<*name*>在表达式中引用分组，用\k<*name*>在替换中引用

⑤Objective-C中用(?<*name*>regex)表示命名分组，用\k<*name*>在表达式中引用分组，在替换中引用的方法未知

⑥Golang中用(?P<*name*>regex)表示命名分组，命名分组的反向引用和在替换中引用的方法未知

（续表）

断言								
特性	.NET	Java	JavaScript	PHP	Python	Ruby	Objective–C	Golang
\b（与\w 规则相同）	√	①	√	√	√	②	③	√
^	√	√	√	√	√	④	√	√
$	√	√	⑤	√	√	④	√	√
\A	√	√	×	√	√	√	√	√
\z	√	√	×	√	×	√	√	√
\Z	√	√	×	√	⑥	√	√	×
(?=*regex*) (?!*regex*)	√	√	√	√	√	√	√	×
(?<=*regex*) (?<!*regex*)	√	⑦	⑧	⑨	⑩	⑪	⑦	×
①Java中的\w采用ASCII匹配规则，但\b采用Unicode匹配规则 ②Ruby 1.9 中的\b采用Unicode匹配规则 ③Objective-C中的\b和\w默认采用Unicode匹配规则，如果指定ASCII匹配规则，受影响的只有\b ④Ruby默认采用多行模式，^和$可以匹配行的起始/结束位置 ⑤JavaScript中的$无法匹配文本末尾行结束符之前的位置 ⑥Python中的\Z等价于其他语言中的\z ⑦逆序环视中的正则表达式能匹配的文本长度必须有上限 ⑧JavaScript必须到ES2017（TC39）之后才支持逆序环视，此时逆序环视的表达式没有限制 ⑨PHP中的逆序环视中的正则表达式匹配文本的长度可以不确定，可以是若干个值，但必须是固定值，多选结构只能出现在顶层 ⑩Python的逆序环视中的正则表达式匹配的文本长度必须是固定的 ⑪Ruby 1.8 不支持逆序环视，Ruby 1.9 的逆序环视中的正则表达式匹配的文本长度必须是固定的								
匹配模式								
特性	.NET	Java	JavaScript	PHP	Python	Ruby	Objective–C	Golang
不区分大小写模式	√	√	√	√	√	√	√	√
单行模式	√	√	√	√	√	√	√	√
多行模式	√	√	√	√	√	①	√	√
注释模式	√	√	×	√	√	√	√	×
(?-*modifier*) 停用模式	√	√	×	√	×	√	√	√
(?*modifier*:*regex*) 模式只对括号内表达式生效	√	√	×	√	×	√	√	√
①Ruby默认采用多行模式								

（续表）

Unicode								
特性	.NET	Java	JavaScript	PHP	Python	Ruby	Objective–C	Golang
Unicode Property	√	√	×	√	×	①	√	√
Unicode Block	√	√	×	×	×	×	×	×
Unicode Script	×	×	×	√	×	①	②	②
①只有Ruby 1.9 支持，使用时需显式指定Unicode模式 ②文档未说明，但测试可用								

附录 B　常用的正则表达式

之前已经讲解了正则表达式的功能和用法，但在日常开发中，需要用正则表达式解决的各种问题经常是固定的、重复的，所以这里列出一些常用的正则表达式供查阅。

因为希望做到尽量通用，所以只列出纯粹的正则表达式（为避免字符组简记法在不同语言中的规定不同，一般尽量不使用字符组简记法。如果确实出现了，除非特殊说明，否则一定是使用**ASCII 匹配规则**，参见第 7 章），这个正则表达式在具体语言中要如何写（如果使用字符串，要如何转义），如何使用（调用哪个 API），都没有介绍，也没有刻意使用非捕获型括号`(?:…)`取代捕获型括号`(…)`，如果你希望了解这些内容，请仔细阅读前面的章节。

另外，如果将正则表达式用于验证，一般应该在表达式首尾加上`\A`和`\z`；如果需要将其用于提取，应该在首尾分别加上对应的断言。这一点，除非特别要注意的地方会明确说明，一般不做说明。

整数

举例

```
1024
89
10
```

正则表达式

```
[0-9]+
```

如果用于数据提取，不需要添加断言。

十六进制数

举例

```
4e00
9F
A3D6
```

正则表达式

```
[0-9a-fA-F]+
```

如果用于数据提取，不需要添加断言。

MD5 字符串

举例

```
C37B58C783456BF879C3DD5C519195A2
44912ED4AD09F0D2CA5066A16209F717
```

正则表达式

```
 [0-9a-fA-F]{32}
```

常见的 MD5 字符串由 32 位十六进制数构成。

如果用于数据提取，还应该在首尾添加断言，保证之前和之后不能出现十六进制字符，也就是在表达式开头加上`(?<![0-9a-fA-F])`，在末尾加上`(?![0-9a-fA-F])`。

浮点数

举例

```
+3.14
+0.07
.7
3.6
-3.14
-12
```

正则表达式

```
(+?([0-9]+|[0-9]+\.[0-9]+|\.[0-9]+)|-?([0-9]+|[0-9]+\.[0-9]+))
```

浮点数包括整数部分、小数点和小数部分，分别由`\d+`、`\.`、`\d+`匹配，单独来看它们都不一定要出现，但不可能三个部分都不出现，所以用多选结构`([0-9]+|[0-9]+\.[0-9]+|\.[0-9]+)`表示。另外，之前可能出现符号，如果出现的是+，则各种形式都没问题，如果出现的是-，一般不会允许出现`-.7`之类的写法，所以还需要用一个多选结构。

如果用于数据提取，还应该在首尾添加断言。一般来说判断之前的字符不是点号，之后的字符一般不能是点号，但如果是句号，则它之后不能是其他数字字符，这样就可以避免错误匹配`2010.12.20`之类的数据。所以要在表达式开头加上`(?<!.)`，在末尾加上`(?!.[0-9])`。

逗号分隔的整数

举例

```
123,456,789
23,456,789
3,456,789
```

正则表达式

```
(?<![0-9])[0-9]{1,3}(,[0-9]{3})*(?![0-9])
```

英文中的数字经常会加入逗号以便识别，规则是从右向左，每 3 位数字为一段，在之前加入一个逗号，比如 12,345,678。最左边一段数字的长度不能超过 3，所以表达式是`[0-9]{1,3}`；其他每段长度都是 3，所以表达式是`,[0-9]{3}`。

因为在这个表达式中必须对各段的长度做出限制，所以首尾应当分别使用断言`(?<![0-9])`和`(?![0-9])`，确保之前和之后没有数字字符。如果你使用的是 Ruby 1.8，不支持逆序环视，也可以用`\b`替代`(?<![0-9])`，作为权宜之计。

中文字符

在 Unicode 编码环境下（详见第 7 章）使用的正则表达式，可匹配 Unicode 字符集中基本的中文字符（足够覆盖常见的绝大多数情况）。

.NET Java JavaScript Python

```
[\u4E00-\u9FFF]
```

如果使用 Python 2 版本，必须在字符串之前添加 u，比如 u"[\u4E00-\u9FFF]"。

PHP

```
[\u{4E00}-\u{9FFF}]
```

必须指定 Unicode 模式，比如/[\u{4E00}-\u{9FFF}]/u。

Ruby

```
[\x{4E00}-\x{9FFF}]
```

必须在 Ruby 1.9 以上版本中使用，而且必须显式指定 Unicode 模式，比如/[\x{4E00}-\x{9FFF}]/u。

在 GBK 编码环境下使用的正则表达式。

Ruby 1.8 PHP Python

```
[\xb0-\xfe][\x00-\xff]
```

这个正则表达式将中文字符作为两个字节加以匹配，所以不需要指定任何与 Unicode 有关的设置，无论是对正则表达式还是字符串，都是这样。而且，如果需要匹配多个中文字符，必须添加括号将两个字节作为一组，再以量词限定，比如`([\xb0-\xfe][\x00-\xff])+`。

中文间多余的空白字符

举例

```
中文的文本，往往会包 含许多这样的空白字符
但是，又不 能简单地将所有空白都删掉，否则you will get a big mess
```

正则表达式

```
(?<=[\u4E00-\u9FFF])\s+(?=[\u4E00-\u9FFF])
(?<=[^\x00-\x7f])\s+(?=[^\x00-\x7f])
```

在填写中文内容时，因为转贴等操作，往往会多出许多空白字符，破坏格式，影响美观。但解决这类问题，又不能简单删掉所有空白字符，否则就破坏了其中的英文文本的格式，所以要删除的只是“中文字符之间的空白字符”（英文以空白字符作为词的分隔，而正常的中文文本之间很少出现空白字符）。如果要处理的文本是 Unicode 编码，则可以直接使用上面介绍的匹配中文字符的表达式`[\u4E00-\u9FFF]`，以逆序环视`(?<=[\u4E00-\u9FFF])`和顺序环视`(?=[\u4E00-\u9FFF])`要求匹配`\s+`的空白字符两侧都必须是中文字符，再将空白字符删除即可。如果要处理的文本不是 Unicode 编码，处理的思路仍然相同，只是假设所有的非 ASCII 字符都是中文字符，用排除型字符组`[^\x00-\x7f]`匹配，这个表达式有点风险，不过大多数情况还是够用的。

因为这个表达式就是用来提取（匹配-替换）的，所以不需要再添加断言。

价格

举例

```
￥189.99
$ 46
88.0
12700
```

正则表达式

```
[￥$]?\s*[0-9]+(\.[0-9]{2})?
```

价格比较好判断，之前可能有货币符号￥或者$，在 Unicode 编码环境中，￥是单个字符，可以直接使用字符组`[￥$]`，否则用`(￥|\$)`比较好，记得`$`必须转义，否则会作为锚点处理。在

这个符号之后，具体的价格数值之间，可能还有若干空白字符，所以要使用`\s*`。如果价格出现了小数部分，一般不会超过 2 位，所以用`(\.[0-9]{1,2})`。

如果用于数据提取，还应该在尾部添加断言，判断之后的字符不是数字字符，也就是在表达式末尾加上`(?![0-9])`。

手机号码

举例

```
1380123467
+861380123467
+86 1380123467
0086 1380123467
01380123467
1827654321
```

正则表达式

```
((0|(00|\+)86)\s?)?(13[0-9]|15[0-356]|18[025-9])\d{8}
```

到本书写作时为止，国内已经开放的手机号段有 130~139、150~153、155~156、180、182、185~189，用多选分支`(13[0-9]|15[0-356]|18[025-9])`可以很准确地匹配号段；之后的 8 位一般没有限制，只要是数字即可，用`\d{8}`匹配。另一方面，手机号之前可能有 `0` 或者`+86` 或者`0086`，而且如果出现，在它和后 11 位之间，可能还有一个空白字符，所以用`(0|(00|\+)86\s?)`匹配，因为这部分是可能出现的，所以还需要加上量词，变为`(0|(00|\+)86\s?)?`。

如果用于数据提取，还应该在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

固定电话号码

举例

```
1234567
021-12345678
0731-12345678
(010)12345678
(010)12345678-836016
```

正则表达式

```
(\(?0[1-9]{2,3}\)?-?)?[1-9][0-9]\{6,7}(-[0-9]{1,6})?
```

国内的电话号码分为两段，区号一般为 3~4 位，且以 0 开头，之后不为 0，所以是`0[1-9]{2,3}`；市内号码一般为 7~8 位，且不能以 0 开头，所以是`[1-9][0-9]\{6,7}`；如果还有分机，分机号一般不超过 6 位，且以-与总机号码相连，所以是`-[0-9]{1,6}`。因为区号可能以括号标注（(010)12345678），也可能以-与市内号码相连（010-12345678），或者直接相连（01012345678），所以区号部分的表达式是`(\(?0[1-9]{2,3} \)?-?)?`；分机号码也不一定出现，所以需要用`?`限定，记为`(-[0-9]{1,6})?`。

如果用于数据提取，还应该在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

邮政编码

举例

```
100859
412000
```

正则表达式

```
[1-9][0-9]{5}
```

邮政编码一般是 6 位数字，首位不能为 0，所以是`[1-9][0-9]{5}`。

这个表达式比较简单，许多数据中都可能出现这样的字符串，所以提取时一定不要忘记在两端添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

身份证号码

举例

```
310100198001012071
100100198001012O7x
310100800101207
```

正则表达式

```
[1-9][0-9]{14}([0-9]{2}[0-9xX])?
```

我国身份证分为两代，第一代身份证号码长为 15 位，第二代身份证号码长为 18 位，都不能以 0 开头。如果是 18 位身份证号，则最后一个字符可以是 x 或者 X（在特定情况下，按照公安系统和网银系统的要求，只能使用大写 X），其他都只能为数字。所以可以这么看，第一位是`[1-9]`，之后是 14 位`[0-9]`，如果是 18 位号码，则最后还会出现`[0-9]{2}[0-9x]`。

如果用于数据提取，还应该在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

QQ 号

举例

```
10000
95381376
```

正则表达式

```
[1-9][0-9]{4,9}
```

目前 QQ 号最长为 10 位，最短为 5 位，且首位不能为 0。

如果用于数据提取，还应该在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

ISBN 号

举例

```
ISBN 978-7-534-74286-6
ISBN-13 9787534742866
ISBN-13: 978-7-534-74286-6
9787534742866
0-596-52068-9
ISBN-10 0-596-52068-9
```

正则表达式

```
((ISBN(-13)?:?\s)?97[89][-\s]?[0-9][-\s]?[0-9]{3}[-\s]?[0-9]{5}[-\s]?[0-9]|(ISBN
(-10)?:?\s)?[0-9][-\s]?[0-9]{3}[-\s]?[0-9]{5}[-\s]?[0-9x])
```

ISBN 有 10 位（ISBN-10）和 13 位（ISBN-13）两种长度，两者的主要区别是：ISBN-10 的开头可能是 `ISBN-10`，而 ISBN-13 的开头只可能是 `ISBN-13`；最后一位是校验码，ISBN-10 中的取值可能是`[0-9x]`，而 ISBN-13 中只可能是`[0-9]`；ISBN-13 的开头 3 位只能是 `978` 或者 `979`。无论 ISBN-10 还是 ISBN-13，最后的 10 位都可以按照 1、3、5、1 分为四段，各段之间可能以-或者空白字符作为分隔符，比如 `7-534-74286-6` 或者 `7 534 74286 6`，或者也可以没有任何分隔符，比如 `7534742866`。

对应开头部分的正则表达式比较容易，ISBN-10 是`(ISBN(-10)?:?\s)?`，ISBN-13 则是

`(ISBN(-13)?:?\s)?`。之后，如果是 ISBN-13，必然会出现 `97[89]`，再将最后 10 位按照 1、3、5、1 四段分开，中间加上可能出现的分隔符`[-\s]?`。如果是 ISBN-10，则最后一位是`[0-9x]`，ISBN-13 则为`[0-9]`，为避免两种情况混淆，将 ISBN-10 和 ISBN-13 两种情况分别作为一个多选分支，列在同一个多选结构内。

如果用于数据提取，还应该在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

IPv4 地址

举例

```
192.168.1.1
202.198.0.68
```

正则表达式

```
(0{0,2}[0-9]|0?[0-9]{2}|1[0-9][0-9]|2[0-4][0-9]|25[0-5]\.){3}(0{0,2}[0-9]|0?[0-9]{2}
|1[0-9][0-9]|2[0-4][0-9]|25[0-5])
```

这个例子在第 3 章讲解过，IPv4 的每一段都在 0~255 之间，为了考虑一位数、两位数、三位数的情况，必须用多选结构来解决。注意，在这个表达式中不能使用反向引用，而应当使用量词。

如果用于数据提取，还应当在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

IPv6 地址

举例

```
fe80:0:0:09a:fe:0:0:4ca2
da8:207:3:21a:64ff:fe6d:d1eb
```

正则表达式

```
([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}
```

IPv6 地址一般分为 8 段，每段都用十六进制数值表示，长度在 1~4 之间，所以是`[0-9a-fA-F]{1,4}`。

如果用于数据提取，还应当在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9a-fA-F])`，在末尾加上`(?![0-9a-fA-F])`。

时间字符串

24 小时制，如 `20:46`

正则表达式

```
(0?[0-9]|1[0-9]|2[0-3]):(0?[0-9]|[1234][0-9]|5[0-9])
```

12 小时制，如 `12:38`

正则表达式

```
(0?[0-9]|1[012]):(0?[0-9]|[1234][0-9]|5[0-9])
```

时间字符串的匹配与之前 IPv4 地址的匹配类似，需要注意的是数据的取值范围：如果是 24 小时制，小时数只能在 0~23 之间，所以正则表达式是`(0?[0-9]|1[0-9]|2[0-3])`；如果是 12 小时制，小时数则只能在 0~12 之间，正则表达式是`(0?[0-9]|1[012])`；分钟数只能在 0~59 之间，所以正则表达式是`(0?[0-9]|[1234][0-9]|5[0-9])`。如果还要包括秒数，则还应该附加`:(0?[0-9]|[1234][0-9]|5[0-9])`。

如果用于数据提取，还应当在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

日期字符串

以 `mm/dd/yyyy` 为例，如 `12/20/2010`

正则表达式

```
(0?[1-9]|1[012])/(0?[1-9]|[12][0-9]|3[01])/[0-9]{4}
```

时间字符串的匹配与之前 IPv4 地址的匹配类似，需要注意的是数据的取值范围：月只能在 0~12 之间，所以正则表达式是`(0?[1-9]|1[012])`；日只能在 0~31 之间（暂不考虑大、小月），所以正则表达式是`(0?[1-9]|[12][0-9]|3[01])`；年一般无限制，所以正则表达式是`[0-9]{4}`。

如果用于数据提取，还应当在首尾添加断言。一般来说，判断之前和之后的字符不是数字字符即可，也就是在表达式开头加上`(?<![0-9])`，在末尾加上`(?![0-9])`。

Windows 路径

举例

```
C:\windows
```

```
e:\Program Files\
d:\starcraft2\startcraft.exe
```

正则表达式

```
[a-zA-Z]:\\([^\\/:*<>|"?\r\n]+\\)*[^\\/:*<>|"\r\n]+\\?
```

Windows 下的路径，开头是盘符，必须是一个英文字母，然后是:\，注意\在正则表达式中必须转义为\\，匹配这部分的正则表达式是`[a-zA-Z]:\\`；文件名（包括目录名）中不能包含\ / : * < > | " ?字符，还不能是换行符，也不能是空字符串，所以表达式是`[^\\/:*<>|"?\r\n]+`，目录之间以\分隔，路径内部的目录名之后必须有\，所以这些部分可以用`([^\\/:*<>|"?\r\n]+\\)*`匹配；路径末尾的文件名如果是目录，可能会以\结尾，所以最后的部分是`[^\\/:*<>|"?\r\n]+\\?`。

如果用于数据提取，还应当在首尾添加断言，在表达式开头加上`(?<![a-z])`，在末尾加上`(?![^\S\\/:*<>|"?])`即可。

UNIX 路径

举例

```
/usr/local
/usr/local/
/usr/bin/python
bin/python
```

正则表达式

```
/?([^\\/:*%<>|"?\r\n]+/)*[^\\/:*%<>|"?\r\n]+/?
```

按照 POSIX 规范，文件名不能包含\ / : * % < > | " ?字符，不能是换行符，也不能是空字符串，所以匹配文件名的表达式是`[^\\/:*%<>|"?\r\n]+`，在路径内部的所有目录名之后必须有/，所以这些部分可以用`[^\\/:*%<>|"?\r\n]+/`匹配；在路径的开头可能有/，所以要加上`/?`；路径末尾的文件名如果是目录，可能以/结尾，所以最后的部分是`[^\\/:*%<>|"?\r\n]+/?`。

如果用于数据提取，还应当在首尾添加断言，一般来说，在表达式开头加上`(?<![^\S\\/:*%<>|"?\r\n])`，在末尾加上`(?![^\S\\/:*%<>|"?\r\n])`。

主机名

举例

```
localhost
```

```
163.com
news.163.com
sina.com.cn
```

正则表达式

```
(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))((?!-)[-a-zA-Z0-9]{1,63}\.)*((?!-)[-a-zA-Z0
-9]){1,63}
```

主机名分为很多个字段（label），每个字段可以包含字母、数字、横线，其长度不能超过 63，所以是`[-a-zA-Z0-9]{1,63}`；且字段不能以横线-开头，这一点用`(?!-)`来保证。总长度不能超过 255，用`(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))`来保证。

整个主机名可以看作“字段+点号”的重复，所以是`((?!-)[-a-zA-Z0-9]{1,63}\.)*`，最后的字段必须要出现，所以最后再添加一个匹配字段的子表达式。

如果用于数据提取，则必须在表达式开头添加断言`(?<![-a-zA-Z0-9.])`。

电子邮件地址

举例

```
someone@localhost
tom@163.com
jerry.lee@sina.com.cn
```

正则表达式

```
(?!.)(?![\w.]*?\.\.)[\w.]{1,64}@(?=[-a-zA-Z0-9.]{0,255}(?![-a-zA-Z0-9.]))((?!-)[-a-z
A-Z0-9]{1,63}\.)*((?!-)[-a-zA-Z0-9]){1,63}
```

电子邮件地址由用户名和主机名两部分构成，以@连接，主机名的匹配之前已经讲过，这里只讲解用户名的匹配。

一般来说，用户名中能出现的字符是字母、数字、下画线、点号，其长度不超过 64 个字符，所以是`[\w.]{1,64}`；用户名不能以点号开头，所以要添加环视`(?!.)`，同时用户名中不能包含连续点号，所以还需要添加环视`(?![\w.]*?\.\.)`。

如果用于数据提取，还必须在之前添加断言`(?<![\w.])`。

URL

举例

```
http://www.yahoo.com
```

```
http://www.yahoo.com/
http://yahoo.com:80
http://news.163.com
http://news.163.com/11/0625/03/77C5GLMJ0001124J.html
http://baike.baidu.com/view/1203256.htm
http://www.baidu.com/s?wd=%D5%FD%D4%F2
```

正则表达式

```
(https?|ftp)://[^/?:]+(:[0-9]{1,5})?(/?|(/[^/]+)*(\?[^\s"']+)?)
```

相对来说，匹配 URL 的表达式比较复杂，因为关于主机名（hostname）、路径名（path）、参数（parameter）等部分都有复杂的规范（所以不少语言中都提供了专门的 URLParse 函数），这里只给出一个相对简单的正则表达式，在日常使用中可以满足需求。

最开始是协议名，可能是 `http`、`https`、`ftp`，这部分对应的表达式是`(https?|ftp)`，然后是`://`；主机名使用了相对简单的`[^/?:]+`匹配（完整的匹配可以参考之前匹配主机名的表达式），然后用`(:[0-9]{1,5})?`匹配可能出现的端口部分（因为端口号最大为 65536，这里简要规定为 5 位数字）。如果这个 URL 只包含主机名（以及可能的端口号），可能到此为止，也可能之后还有/（比如 `http://www.yahoo.com/`）；如果之后还有内容，一般是以/分隔开的路径名，所以用`(/[^/]+)*`匹配，最后可能还包括参数（比如`?wd=%D5%FD%D4%F2`），用`(\?[^\s"']+)?`匹配（这里假定 URL 中不包含空字符和引号字符，而且 URL 之后必然有空字符，或者位于字符串的末尾）。

URL 的规范比较多，这个表达式能进行一般的验证，也能在相对规范（通常的文档）的数据中提取出 URL，如果存在很多噪声数据（比如似是而非的 URL），这个表达式可能无能为力，此时请参考其他资料。

HTML Entity

举例

```
&#x53d1;
&#21457;
```

正则表达式

```
&#([0-9]+|x[0-9a-fA-F]+);
```

HTML Entity 是 HTML 代码中常见的数据，格式为&#x*hex*;或者&#*dec*;，其中的 *hex* 或者 *dec* 为所示字符的码值，*hex* 为十六进制，*dec* 为十进制，所以`发`和`发`其实都表示 Unicode 编码的“发”。表达式开头的&# 和结尾的;都不难理解，其中的多选结构，`[0-9]+`和`x[0-9a-fA-F]+`分别用来匹配十进制和十六进制数值的字符串。

如果用于数据提取，不需要添加断言。

HTML tag

举例

```
<a href="http://www.yahoo.com">
<img src="pic.jpg"/>
</h1>
```

正则表达式

```
(?i)</?[a-z][-a-z0-9_:.]*(?=[\s>])('[^']*'|"[^"]*"|[^'">])*>
```

按照规范，HTML tag 必须有 tag name，第一个字符（close tag 中是/之后的字符，为了应对 close tag，在开头加上`/?`）必须是字母`[a-z]`，其中可以包含的字符还有数字`[0-9]`、横线-、下画线_、冒号:、句号.，所以用`[a-z][-a-z0-9_:.]*`匹配 tag name，为保证它完整匹配 tag name，用肯定顺序环视`(?=[\s>])`保证它之后是空白字符或者>。在结尾的>之前可能出现的文本，只有在单引号字符串`'[^']*'`或双引号字符串`"[^"]*"`内部可能出现>，否则不能出现>，这部分内容用`[^'">]`匹配，最后是结尾的>。同时，这个表达式指定了不区分大小写模式`(?i)`，保证 tag name 可以为大写。

如果用于数据提取，不用添加断言。

Title tag

举例

```
<title>标题</title>
<TITEL>网站标题</TITLE>
<Title>标题</Title>
```

正则表达式

```
(?i)<title>.*?</title>
```

这个表达式比较简单，不多解释，只是要注意使用不区分大小写模式应对各种可能的情况。

成对的 tag

举例

```
<h1>yahoo</h1>
```

```
<span>yahoo</span>
<div><img src="pic.jpg" /></div>
```

正则表达式

```
(?is)<([a-z][-a-z0-9_:.]*)(?=[\s>])('[^']*'|"[^"]*"|[^'">])*>.*?</\1>
```

这个正则表达式与上一个很类似，只是给匹配 tag name 的表达式加上一个捕获型括号，并在表达式结尾以`</\1>`引用对应的 close tag。两个 tag 之间的内容用`.*?`匹配，注意必须使用忽略优先量词`*?`，并且必须使用单行模式（为了照顾 tag name 的大写情况，同时指定了不区分大小写模式）`(?is)`。因为 JavaScript 不支持单行模式，所以在 JavaScript 中必须将`.*?`改为`[\s\S]*?`，将`(?is)`改为`(?i)`。

如果用于数据提取，不用添加断言。

img tag

举例

```
<img src="pic.jpg"/>
<img src='pic.jpg'/>
<img src = pic.jpg />
<img width="1024" height="768" src="pic.jpg" class="do-edit"/>
```

正则表达式

```
(?i)<img\s('[^']*'|"[^"]*"|[^'">])*src\s*=\s*['"]?[^'"\s]+['"]?('[^']*'|"[^"]*"|[^'">])*/>
```

img tag 以 img 开头，而且之后至少有一个空白字符，用`\s`匹配，然后用之前讲过的`('[^']*'|"[^"]*"|[^'">])*`匹配 tag 内 src 之前可能出现的内容，用`src\s*=\s*['"]?[^'"\s]+['"]?`匹配图片的地址，最后再用`('[^']*'|"[^"]*"|[^'">])*`匹配其他可能的内容，最后是`/>`。注意这个表达式同样要使用`(?i)`指定不区分大小写模式。

如果用于数据提取，可以把`src\s*=\s*['"]?[^'"\s]+['"]?`改为`src\s*=\s*['"]?([^'"\s]+)['"]?`，即加上捕获型括号，匹配成功之后提取编号为 2 的分组即可。

table tag

举例

```
<table><tr><td>1</td></tr></table>
<table border="1"><tr><td>1</td></tr></table>
```

正则表达式

```
(?is)<table\s(?=[\s>])('[^']*'|"[^"]*"|[^'">])*>.*?</table>
```

table tag 以 table 开头，之后可能是空白字符也可能是>，所以用`(?=[\s>])`，然后用之前讲过的`('[^']*'|"[^"]*"|[^'">])*`匹配 tag 内可能出现的内容，再往后是`>`。close tag 用`</table>`匹配，之间的内容用`.*?`匹配，注意必须使用忽略优先量词`*?`，并且必须使用单行模式（为了照顾 tag name 的大写情况，同时指定了不区分大小写模式）`(?is)`。因为 JavaScript 不支持单行模式，所以在 JavaScript 中必须将`.*?`改为`[\s\S]*?`。

如果用于数据提取，一般还需要给`.*?`加上捕获型括号变为`(.*?)`，匹配成功之后用分组 2 提取出 table 的内容，再按照 tr tag 拆分成行，每一行按照 td 拆分成列，得到单元格的内容，匹配 tr tag 和 td tag 的正则表达式与 table tag 的非常相似，此处不重复。

href tag

举例

```
<a href="http://www.yahoo.com">yahoo</a>
<a href='http://www.yahoo.com'>yahoo</a>
<a href = http://www.yahoo.com>yahoo</a>
<a id="1" href="http://www.yahoo.com" class="do-edit">yahoo</a>
<a href="http://www.yahoo.com"><img src="pic.jpg" /></a>
```

正则表达式

```
(?i)<a\s('[^']*'|"[^"]*"|[^'">])*href\s*=\s*['"]?[^'"\s]+['"]?('[^']*'|"[^"]*"|[^'">
])*/>.+?</a>
```

这个表达式类似匹配 img tag 的表达式，只是将 src 属性改为 href 属性，另外加上了匹配 close tag 的`</a>`。在 open tag 和 close tag 之间的内容用`.+?`匹配，注意不能用`[^<]+?`，因为其中还有其他 tag，比如`<a …><img …></a>`，另外要使用忽略优先量词`+?`，防止错误匹配。

如果用于数据提取，不用添加断言，但是应该给`[^'"\s]+`和`.+?`分别加上捕获型括号，匹配成功之后用分组 1 提取链接的地址，用分组 3 提取链接的文本。

JavaScript 代码

举例

```
<script type="text/javascript">alert("1")</script>
<script type="applicate/javascript">alert("1")</script>
<script language="text/javascript">alert("1")</script>
<script>alert("1")</script>
```

正则表达式

```
(?is)<script[\s>].+?</script>
```

JavaScript代码在`<script`之后可能并不是空格，而是空白字符，再之后，可能是`type="text/`

javascript"，也可能是type="application/javascript"，还可能用language取代type（实际上language是以前的写法，现在大都用type），甚至可能没有属性，直接是<script>。[1] 所以这个表达式应该改造一下，不妨将条件放宽，在script之后，可能出现空白字符，也可能直接是>，可以用一个字符组[\s>]来匹配，之后的内容统一用.+?匹配，忽略优先量词保证了它会匹配到</script>结束。最终得到的表达式就是<script[\s>].+?</script>。注意，这个表达式要用(?is)指定不区分大小写模式和单行模式，前者保证可以兼容<SCRIPT>的情况，后者保证.+?匹配的文本可以跨越多行。因为JavaScript语言中的正则表达式不支持单行模式，所以如果使用在JavaScript程序中，应当将.+?改为[\s\S]+?，将(?is)改为(?i)。

与上面几个匹配 tag 的表达式相比，似乎不够严谨，但实际使用中并不会有太大的问题，因为提取各种 tag 时，更感兴趣的是 open tag 和 close tag 之内的内容而不是 tag 本身，而提取 JavaScript 代码时，一般提取整段即可。

如果用于数据提取，不用添加断言。

XML 格式注释

举例

```
<!--some -->
```

正则表达式

```
(?s)<!--.*?-->
```

XML 文档中的注释以<!--开头，以-->结束。因为 JavaScript 不支持单行模式，所以在 JavaScript 中必须将.*?改为[\s\S]*?，将(?is)改为(?i)。

如果用于数据提取，不用添加断言。

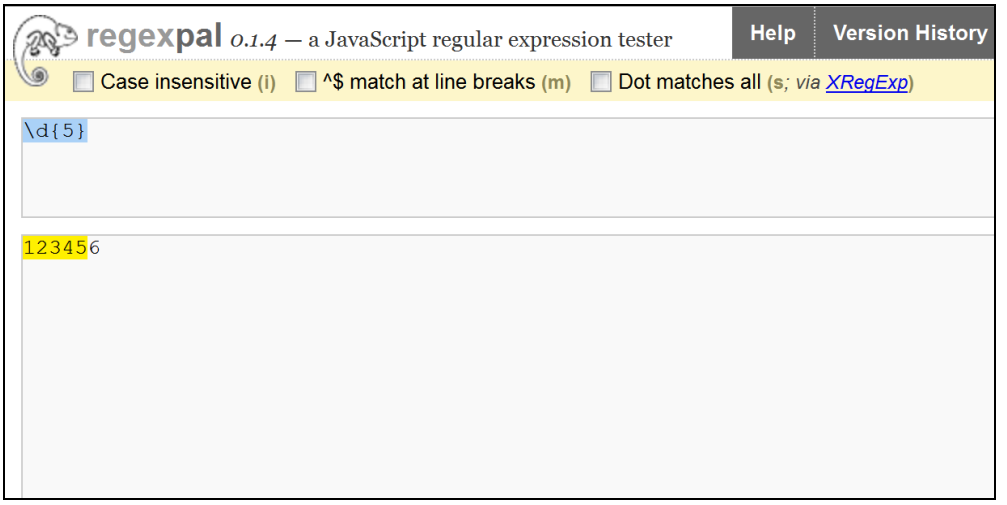
[1] 严格说来，如果只出现<script>，无法保证这里出现的就是 JavaScript 代码，也可能是 VBScript 代码。但考虑到真实世界中的情况，可以认为<script>标识的就是 JavaScript 代码，所以这里不进行区分。

附录 C　常用的正则表达式工具及资源

在线类

RegexPal

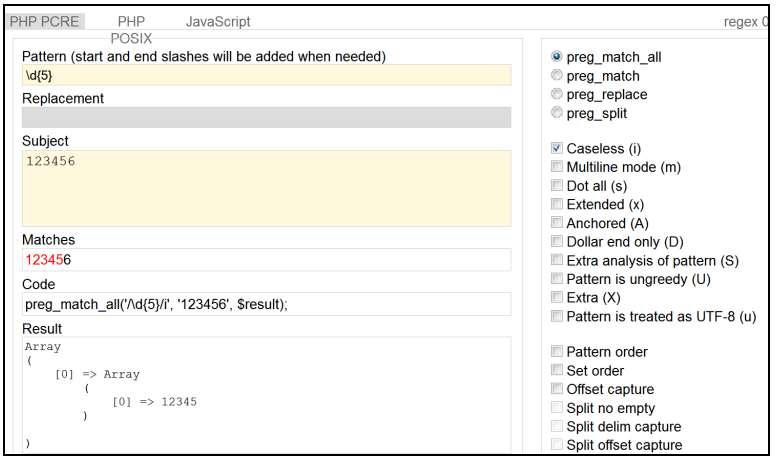
http://www.regexpal.com



这个网站可以测试 JavaScript 的正则表达式，在上面的文本框中输入正则表达式（不需要带上两端的分隔符），在下面的文本框中输入要处理的文本，运行结果会以高亮文字实时标注出来；各个匹配模式列在最上端，可以勾选。它主要进行正则表达式的验证和匹配，不能进行替换和切分。

Larsolavtorvik

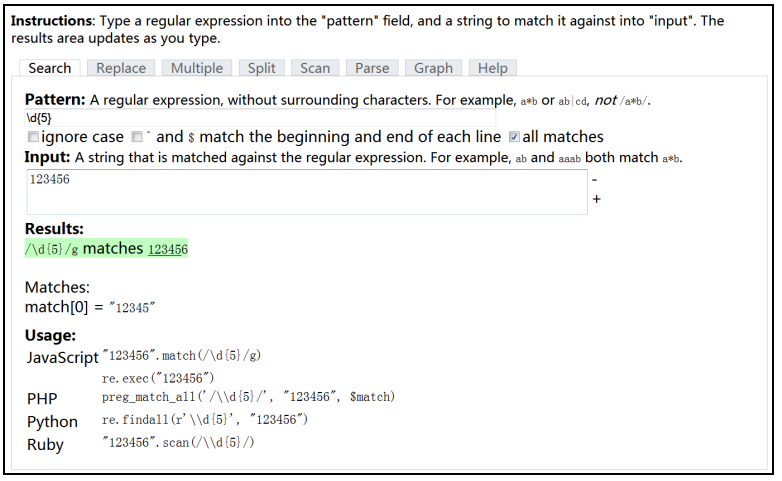
http://regex.larsolavtorvik.com/



这个网站用来调试 PHP 和 JavaScript 的正则表达式，它的功能比 regexpal.com 强大很多。以 PHP 为例，先在左上角选择 PHP PCRE（常用的 preg 系列函数也就是默认的 PHP PCRE 选项），之后页面上会出现若干文本框。在 Pattern 中输入正则表达式（不需要带上两端的分隔符），在 Replacement 中可以输入 replacement 字符串，在 Subject 中输入要处理的文本，在 Code 栏中会自动生成 PHP 代码，在 Result 栏中会显示程序的运行结果。在右侧提供了各种函数、匹配模式、flags 的选择，可以勾选。

ReWork

http://osteele.com/tools/rework/



这个网站同时提供了 JavaScript、Java、Python、Ruby 的正则表达式调试功能，在其中可以进行查找、替换、切分等多种操作，并且在 Mutiple 选项卡中可以同时显示对多个字符串的处理

结果，方便对比（但是它不支持环视）。在 Pattern 栏输入正则表达式，在 Input 栏输入要处理的字符串，可以在 Pattern 栏下勾选匹配模式对应的选项。之后会在 Results 栏实时高亮标注匹配结果。

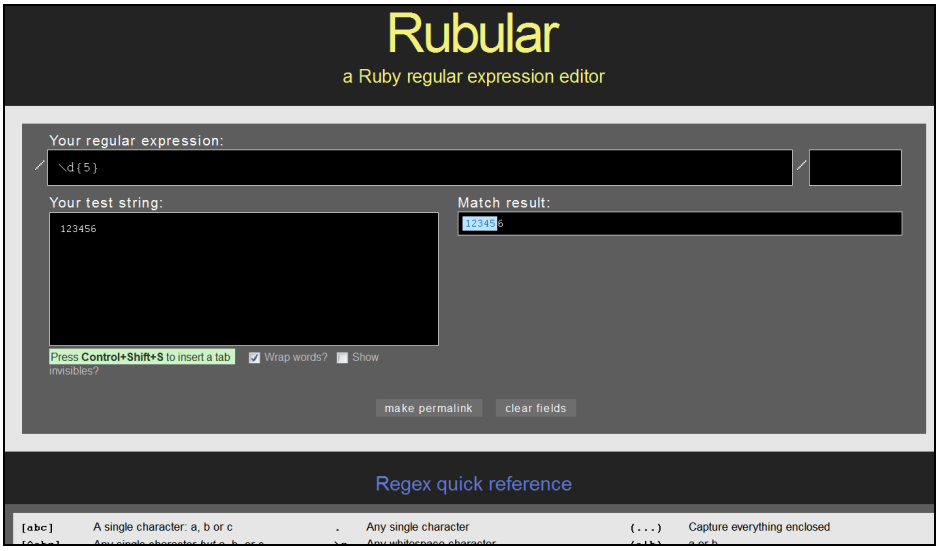
Nregex

http://nregex.com/nregex/default.aspx

这个网站用来调试.NET（C#）中的正则表达式。在 Regular Expression 栏输入正则表达式（如果要设定匹配模式，可以勾选下方对应的选项），在最下面的文本框中输入要处理的字符串（也可以选择从文件或者 URL 导入）；如果要进行替换，可以在 Replacement String 中输入 replacement 字符串。之后，就会在 Matches & Replacements 栏显示程序运行的结果，它右侧的 Matched Groups 中详细显示了匹配的细节。

Rubular

http://www.rubular.com

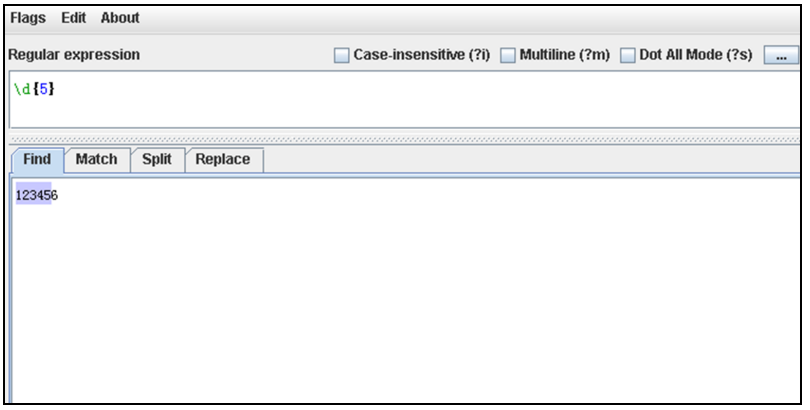


这个网站用来调试 Ruby 1.8 中的正则表达式。它非常形象，最上面的两个文本框，即两个分隔符/之间的文本框用来输入正则表达式，第二个分隔符/右侧的文本框用来指定匹配模式；将要处理的文本输入 Your test string 文本框，右侧的 Match result 中会高亮标注出匹配结果。

因为这个网站只支持 Ruby 1.8，所以许多 Ruby 1.9 才提供的功能（比如逆序环视）都不能使用。

MyRegexp

http://myregexp.com/



这个网站用来调试 Java 中的正则表达式（使用这个网站的调试功能，需要使用 Applet，要求本机有 JRE）。在 Regular expression 栏输入正则表达式，如果需要设定匹配模式，再选择下面的

Find、Match、Split、Replace 四个选项卡，可以进行对应的操作。如果要设定匹配模式，可以勾选右侧的选项，也可以从 Flags 下拉菜单中选择；Edit 下拉菜单中提供了几个很有用的功能：将当前表达式转换为 Java 源代码、XML 源代码、JavaScript 源代码，也可以从 Java 源代码、XML 源代码、JavaScript 源代码获得正则表达式，这样就省去了繁杂的转义。

RegExLib

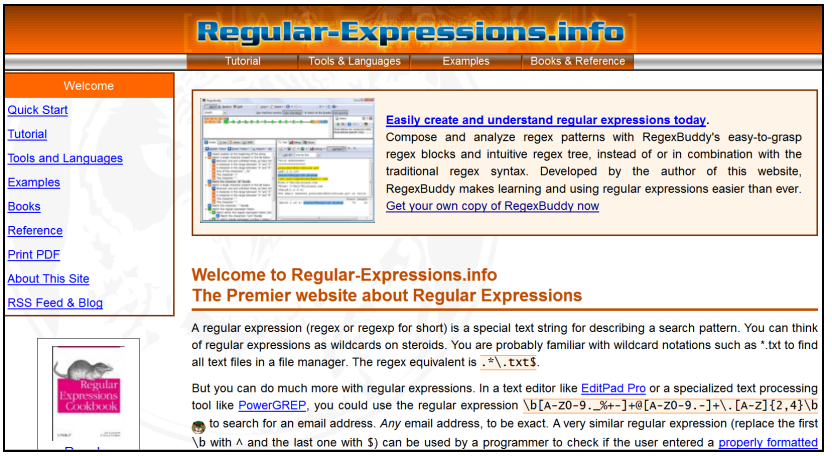
http://www.regexlib.com/

RegExLib.com
Regular Expression Library
^[a-zA-Z]+$
Home
Search
Regex Tester
Browse Expressions
Add Regex
Login
Please support RegExLib Sponsors
Subscribe
Recent Expressions
190 readers
Site Links
Regex Cheat Sheet
Search
Regex Tester
Browse Expressions
Add Regex
Manage My Expressions
Contributors
Welcome to RegExLib.com, the Internet's first Regular Expression Library. Currently we have indexed 3182 expressions from 1878 contributors around the world. We hope you'll find this site useful and come back whenever you need help writing an expression, you're looking for an expression for a particular task, or are ready to contribute new expressions you've just figured out. Thanks!
Add Regex
Find Expressions
Enter Keywords (e.g. email)
Search
Advanced Search
Product Spotlight
Web 2.0 apps in minutes!
Generate data entry and reporting .NET Web apps in minutes, straight from your database. Visually stunning, easy to customize and ready to deploy. Download Now!
Sponsors
Sencha

这个网站堪称正则表达式的知识大全。它内含了许多关于正则表达式的讨论和总结，首页的 Search 功能可以找到与各种任务（比如 E-mail、URL、password）相关的正则表达式，Browse Expressions 则可以浏览网站内总结的表达式，并且提供 Email/Uri/Numbers/Strings/Dates and Times 等诸多分类；另外值得一提的是 Regex Tester，其中提供了针对.NET 和 Silverlight 的正则表达式测试，在其中可以选择各种正则引擎（.NET、Client-side、Silverlight），在 Source 栏输入要处理的文本（也可以从外部文件或者 URL 导入），在 Regular Expression 栏输入正则表达式，单击 Submit 按钮，则会出现匹配结果。

Regular-Expressions

http://www.regular-expressions.info/



这个网站同样囊括了与正则表达式相关的各种知识，它比 regexlib.com 更加全面，其中不但有知识点的讲解，还有各语言的实现细节参考列表，并给出了许多例子，非常适合参考和引用。

软件类

Expresso

http://www.ultrapico.com/Expresso.htm

这是.NET 下的一个正则表达式测试工具，它能显示匹配的细节，方便确定问题所在，推荐使用。这个软件需要注册，但注册是免费的。

RegexBuddy

http://www.regexbuddy.com

这是 Windows 下关于正则表达式的一个很完善的工具，它可以很方便地调试正则表达式，显示各部分匹配的细节。依靠这些功能，很容易理解复杂的表达式。它还可以自动为各种编程语言生成对应的代码片段，非常实用。这个软件需要注册使用，收费 39.95 美元。

PowerGREP

http://www.powergrep.com

PowerGREP 堪称 Windows 下最全面、最强大的正则表达式工具，不但可以调试各种正则表达式，还可以用正则表达式完成各种文本编辑、文件改名等任务。这个软件需要注册使用，收费 159 美元。

JFLAP

http://www.cs.duke.edu/csed/jflap/

JFLAP 是杜克大学计算机系提供的，用来学习形式语言及自动机理论的工具，可以很方便地创建 DFA、NFA、正则语法、正则表达式，也可以进行正规语言、正则表达式、自动机之间的转换，最重要的是，它可以很形象地以图形说明原理和过程。JFLAP 以 jar 文件的方式提供，运行它要求本地有 JDK。

RegexUtil

http://sourceforge.net/projects/regex-util/

这个软件是之前提到的 myregexp.com 提供的，它是 Eclipse 的一个插件，用来调试 Java 的正则表达式。安装之后，在 Eclipse 中选择 Window→Show View→Other→Regex Util 即可运行。

正则表达式术语中英文对照表

虽然本书全部用中文写成，但考虑到英文资料的丰富性，还是建议大家能懂得对应概念的英文表达，方便搜索和进一步学习。

中文	英文	举例	说明
字符组	Character Class	[a-z]	也有资料翻译为“字符集”
字符组简记法	Character Class Shortcuts	\d \D	
排除型字符组	Negative Character Class	[^aeiou]	也有资料翻译为“否定型字符组”
量词	Quantifier	*　? +	
匹配优先量词	Greedy Quantifier	*+ ?+ ++	也有资料翻译为“贪婪量词”
忽略优先量词	Lazy Quantifier	** ?* +*	也有资料翻译为“懒惰量词”，英文也作 Reluctant Quantifier
转义序列	Escape Sequence	\u001B	
字符串文字	String Literal		
原文引用	Quotation	\Q…\E	
括号分组	Grouping	(…)	
固化分组	Atomic Group		
多选结构	Alternative	(*regex1*|*regex2*)	
多选分支	Option		
捕获分组	Capturing Group	(*to be captured*)	
非捕获分组	Non-Capturing Group	(?:*not captured*)	
反向引用	Back-Reference	\1 \2	
命名分组	Named Group		各语言中写法不同
断言	Assertion		有时也写作 Boundary
单词边界	Word Boundary	\b	
锚点	Anchor	^ $	
环视	Look Around		

（续表）

中文	英文	举例	说明
肯定顺序环视	Positive Lookahead	(?=*regex*)	
肯定逆序环视	Positive Lookbehind	(?<=*regex*)	
否定顺序环视	Negative Lookahead	(?!*regex*)	
否定逆序环视	Negative Lookbehind	(?<!*regex*)	
匹配模式	Match Mode		有时也写作 Options
不区分大消息模式	Case-Insensitive		
单行模式	Dot-Matchall		
多行模式	Multiline		
模式修饰符	Modifier		